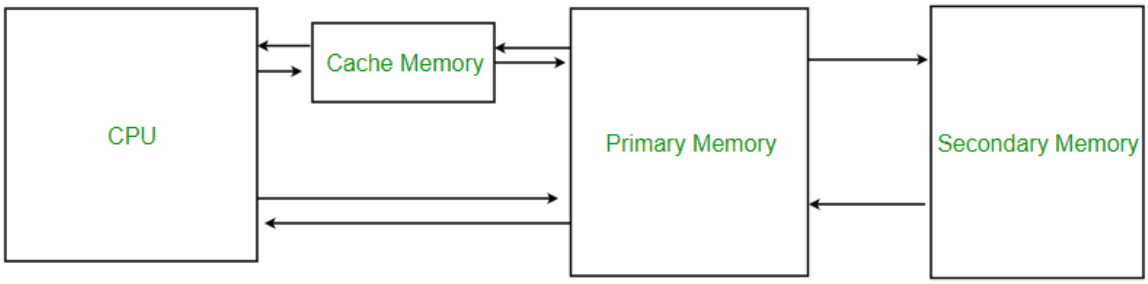
Cache Memory in Computer Organization

Cache Memory is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.
Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instructions and data.



Levels of memory:

- **Level 1 or Register –**
  It is a type of memory in which data is stored and accepted that are immediately stored in CPU. Most commonly used register is accumulator, Program counter, address register etc.
- **Level 2 or Cache memory –**
  It is the fastest memory which has faster access time where data is temporarily stored for faster access.
- **Level 3 or Main Memory –**
  It is memory on which computer works currently. It is small in size and once power is off data no longer stays in this memory.
- **Level 4 or Secondary Memory –**
  It is external memory which is not as fast as main memory but data stays permanently in this memory.

Cache Performance:
When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache.

- If the processor finds that the memory location is in the cache, a **cache hit** has occurred and data is read from cache
- If the processor **does not** find the memory location in the cache, a **cache miss** has occurred. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.

The performance of cache memory is frequently measured in terms of a quantity called **Hit ratio.**

```
Hit ratio = hit / (hit + miss) =  no. of hits/total accesses
```

We can improve Cache performance using higher cache block size, higher associativity, reduce miss rate, reduce miss penalty, and reduce Reduce the time to hit in the cache.

Cache Mapping:

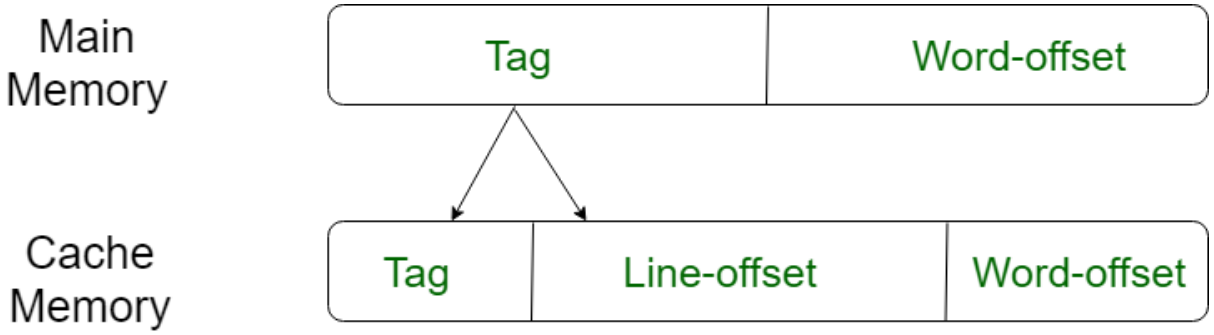
There are three different types of mapping used for the purpose of cache memory which are as follows: Direct mapping, Associative mapping, and Set-Associative mapping. These are explained below.

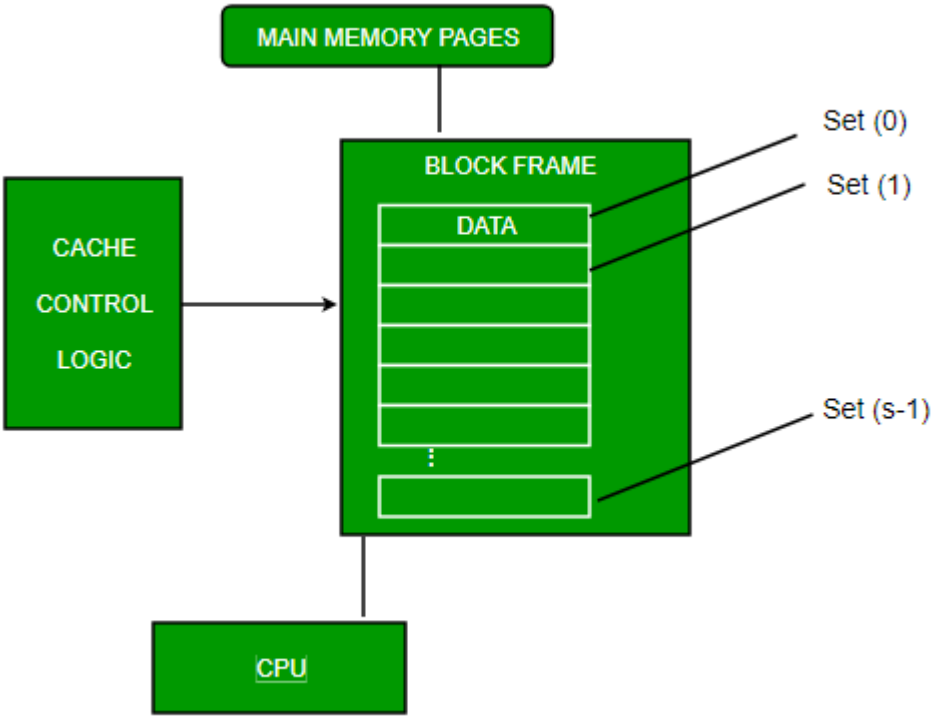
1. **Direct Mapping –**
   The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. or
   In Direct mapping, assigne each memory block to a specific line in the cache. If a line is previously taken up by a memory block when a new block needs to be loaded, the old block is trashed. An address space is split into two parts index field and a tag field. The cache is used to store the tag field whereas the rest is stored in the main memory. Direct mapping`s performance is directly proportional to the Hit ratio.

```
2. i = j modulo m

3. where

4. i=cache line number

5. j= main memory block number
```

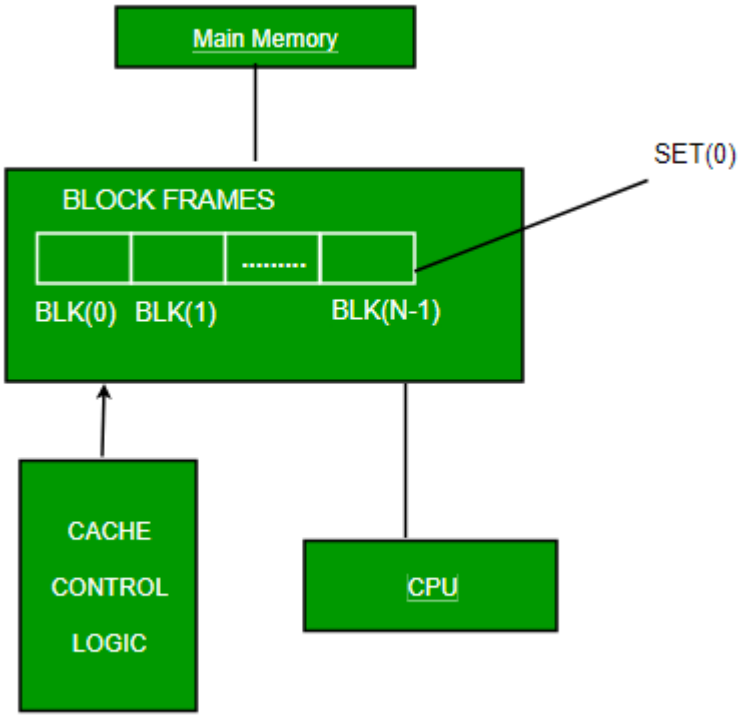
```
m=number of lines in the cache
```

For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory. In most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of s-r bits (most significant portion) and a line field of r bits. This latter field identifies one of the $m=2^r$ lines of the cache.



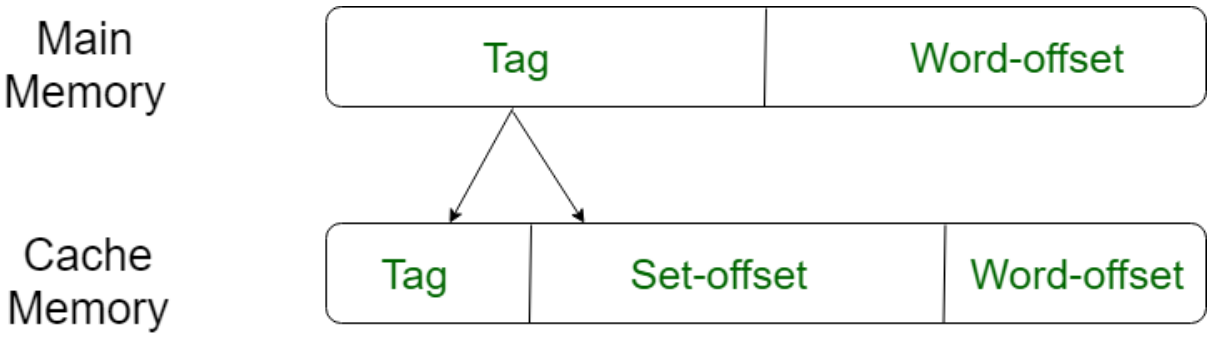


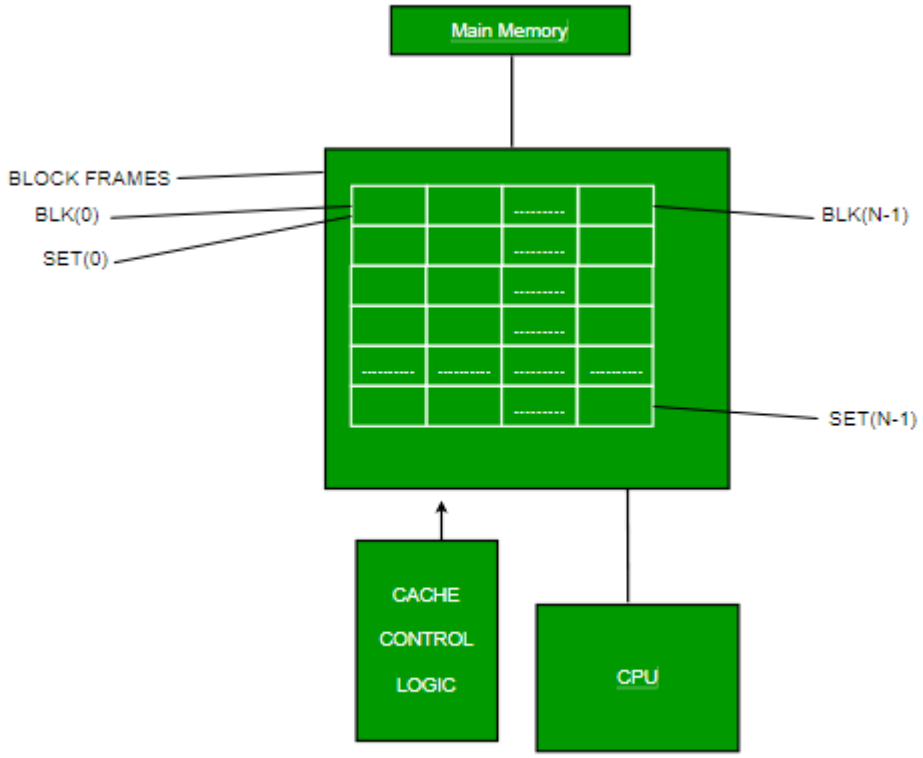
6. **Associative Mapping –**
   In this type of mapping, the associative memory is used to store content and addresses of the memory word. Any block can go into any line of the cache. This means that the word id bits are used to identify which word in the block is needed, but the tag becomes all of the remaining bits. This enables the placement of any word at any place in the cache memory. It is considered to be the fastest and the most flexible mapping form.



7. **Set-associative Mapping –**
   This form of mapping is an enhanced form of direct mapping where the drawbacks of direct mapping are removed. Set associative addresses the problem of possible thrashing in the direct mapping method. It does this by saying that instead of having exactly one line that a block can map to in the cache, we will group a few lines together creating a ***set***. Then a block in memory can map to any one of the lines of a specific set..Set-associative mapping allows that each word that is present in the cache can have two or more words in the main memory for the same index address. Set associative cache mapping combines the best of direct and associative cache mapping techniques.
   In this case, the cache consists of a number of sets, each of which consists of a number of lines. The relationships are

```
m = v * k

i= j mod v


where

i=cache set number

j=main memory block number

v=number of sets

m=number of lines in the cache number of sets

k=number of lines in each set
```





Application of Cache Memory –

1. Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory.
2. The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

Types of Cache –

- **Primary Cache –**
  A primary cache is always located on the processor chip. This cache is small and its access time is comparable to that of processor registers.
- **Secondary Cache –**
  Secondary cache is placed between the primary cache and the rest of the memory. It is referred to as the level 2 (L2) cache. Often, the Level 2 cache is also housed on the processor chip.

Locality of reference –
Since size of cache memory is less as compared to main memory. So to check which part of main memory should be given priority and loaded in cache is decided based on locality of reference.

Types of Locality of reference

5. **Spatial Locality of reference**
   This says that there is a chance that element will be present in the close proximity to the reference point and next time if again searched then more close proximity to the point of reference.
6. **Temporal Locality of reference**
   In this Least recently used algorithm will be used. Whenever there is page fault occurs within a word will not only load word in main memory but complete page fault will be loaded because spatial locality of reference rule says that if you are referring any word next word will be referred in its register that's why we load complete page table so the complete block will be loaded.

Que-1: A computer has a 256 KByte, 4-way set associative, write back data cache with the block size of 32 Bytes. The processor sends 32-bit addresses to the cache controller. Each cache tag directory entry contains, in addition, to address tag, 2 valid bits, 1 modified bit and 1 replacement bit. The number of bits in the tag field of an address is

(A) 11

(B) 14

(C) 16

(D) 27

Answer: (C)

A set-associative scheme is a hybrid between a fully associative cache, and direct mapped cache. It's considered a reasonable compromise between the complex hardware needed for fully associative caches (which requires parallel searches of all slots), and the simplistic direct-mapped scheme, which may cause collisions of addresses to the same slot (similar to collisions in a hash table).

Number of blocks = Cache-Size/Block-Size = 256 KB / 32 Bytes = 2^{13}
Number of Sets = 2^{13} / 4 = 2^{11}
Tag + Set offset + Byte offset = 32
Tag + 11 + 5 = 32
Tag = 16

ROUND ROBIN SCHEDULING

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way.

- It is simple, easy to implement, and starvation-free as all processes get fair share of CPU.
- One of the most commonly used technique in CPU scheduling as a core.
- It is preemptive as processes are assigned CPU only for a fixed slice of time at most.
- The disadvantage of it is more overhead of context switching.

SERIAL NO.	ADVANTAGES	DISADVANTAGES
1.	There is fairness since every process gets equal share of CPU.	There is Larger waiting time and Response time.
2.	The newly created process is added to end of ready queue.	There is Low throughput.
3.	A round-robin scheduler generally employs time-sharing, giving each job a time slot or quantum.	There is Context Switches.
4.	While performing a round-robin scheduling,a particular time quantum is alloted to different jobs.	Gantt chart seems to come too big (if quantum time is less for scheduling.For Example:1 ms for big scheduling.)
5.	Each process get a chance to reschedule after a particular quantum time in this scheduling.	Time consuming scheduling for small quantums .

Illustration:

Round Robin Example:

Process	Duration	Order	Arrival Time
P1	3	1	0
P2	4	2	0
P3	3	3	0

Suppose time quantum is 1 unit.

P1	P2	P3	P1	P2	P3	P1	P2	P3	P2

0 10

P1 waiting time : 4

The average waiting time(AWT) : (4+6+6)/3=5.33

P2 waiting time: 6

P3 waiting time: 6

How to compute below times in Round Robin using a program?

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time. Waiting Time = Turn Around Time – Burst Time

In this post, we have assumed arrival times as 0, so turn around and completion times are same.

The tricky part is to compute waiting times. Once waiting times are computed, turn around times can be quickly computed.

Steps to find waiting times of all processes:

```
1- Create an array rem_bt[] to keep track of remaining
   burst time of processes. This array is initially a
   copy of bt[] (burst times array)
2- Create another array wt[] to store waiting times
   of processes. Initialize this array as 0.
3- Initialize time : t = 0
4- Keep traversing the all processes while all processes
   are not done. Do following for i'th process if it is
   not done yet.
    a- If rem_bt[i] > quantum
       (i)  t = t + quantum
       (ii) bt_rem[i] -= quantum;
    c- Else // Last cycle for this process
```

```
       (i)  t = t + bt_rem[i];
       (ii) wt[i] = t - bt[i]
       (ii) bt_rem[i] = 0; // This process is over
```

Once we have waiting times, we can compute turn around time tat[i] of a process as sum of waiting and burst times, i.e., wt[i] + bt[i]

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

// Function to find the waiting time for all
// processes
void findWaitingTime(int processes[], int n,
             int bt[], int wt[], int quantum)
{
    // Make a copy of burst times bt[] to store remaining
    // burst times.
    int rem_bt[n];
    for (int i = 0 ; i < n ; i++)
        rem_bt[i] =  bt[i];

    int t = 0; // Current time

    // Keep traversing processes in round robin manner
    // until all of them are not done.
    while (1)
    {
        bool done = true;

        // Traverse all processes one by one repeatedly
        for (int i = 0 ; i < n; i++)
        {
            // If burst time of a process is greater than 0
            // then only need to process further
            if (rem_bt[i] > 0)
            {
                done = false; // There is a pending process

                if (rem_bt[i] > quantum)
                {
                    // Increase the value of t i.e. shows
                    // how much time a process has been processed
                    t += quantum;

                    // Decrease the burst_time of current process
                    // by quantum
                    rem_bt[i] -= quantum;
                }

                // If burst time is smaller than or equal to
                // quantum. Last cycle for this process
                else
                {
                    // Increase the value of t i.e. shows
                    // how much time a process has been processed
                    t = t + rem_bt[i];

                    // Waiting time is current time minus time
```

```
                    // used by this process
                    wt[i] = t - bt[i];

                    // As the process gets fully executed
                    // make its remaining burst time = 0
                    rem_bt[i] = 0;
                }
            }
        }

        // If all processes are done
        if (done == true)
          break;
    }
}

// Function to calculate turn around time
void findTurnAroundTime(int processes[], int n,
                        int bt[], int wt[], int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int i = 0; i < n ; i++)
        tat[i] = bt[i] + wt[i];
}

// Function to calculate average time
void findavgTime(int processes[], int n, int bt[],
                                      int quantum)
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    // Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt, quantum);

    // Function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);

    // Display processes along with all details
    cout << "Processes "<< " Burst time "
         << " Waiting time " << " Turn around time\n";

    // Calculate total waiting time and total turn
    // around time
    for (int i=0; i<n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << " " << i+1 << "\t\t" << bt[i] <<"\t "
             << wt[i] <<"\t\t " << tat[i] <<endl;
    }

    cout << "Average waiting time = "
         << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
         << (float)total_tat / (float)n;
}

// Driver code
```

```
int main()
{
    // process id's
    int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];

    // Burst time of all processes
    int burst_time[] = {10, 5, 8};

    // Time quantum
    int quantum = 2;
    findavgTime(processes, n, burst_time, quantum);
    return 0;
}
```

Output:

```
Processes  Burst time  Waiting time  Turn around time

 1         10          13           23

 2         5          10           15

 3         8          13           21

Average waiting time = 12

Average turn around time = 19.6667
```

HASHMAP INTERNALS

HashMap in Java works on hashing principle. It is a data structure which allows us to store object and retrieve it in constant time `O(1)` provided we know the key. In hashing, hash functions are used to link key and value in HashMap. Objects are stored by calling `put(key, value)` method of HashMap and retrieved by calling `get(key)` method. When we call put method, hashcode() method of the key object is called so that hash function of the map can find a bucket location to store value object, which is actually an index of the internal array, known as the table. HashMap internally stores mapping in the form of **`Map.Entry`** object which contains both key and value object. When you want to retrieve the object, you call the get() method and again pass the key object. This time again key object generate same hash code (it's mandatory for it to do so to retrieve the object and that's why HashMap keys are immutable e.g. String) and we end up at same bucket location. If there is only one object then it is returned and that's your value object which you have stored earlier. Things get little tricky when collisions occur. It's easy to answer this question if you have read good book or course on data structure and algorithms like this one. If you know how hash table data structure works then this is a piece of cake.

Since the internal array of HashMap is of fixed size, and if you keep storing objects, at some point of time hash function will return same bucket location for two different keys, this is called collision in HashMap. In this case, a linked list is

formed at that bucket location and a new entry is stored as next node.

If we try to retrieve an object from this linked list, we need an extra check to search correct value, this is done by equals() method. Since each node contains an entry, HashMap keeps comparing entry's key object with the passed key using equals() and when it return true, Map returns the corresponding value.

Since searching inlined list is O(n) operation, in worst case hash collision reduce a map to linked list. This issue is recently addressed in Java 8 by replacing linked list to the tree to search in `O(logN)` time. By the way, you can easily verify how HashMap works by looking at the code of `HashMap.java` in your Eclipse IDE if you know how to attach source code of JDK in Eclipse.

How HashMap works in Java or sometimes how does get method work in HashMap is a very common question on Java interviews nowadays. Almost everybody who worked in Java knows about HashMap, where to use HashMap and difference between Hashtable and HashMap then why this interview question becomes so special? Because of the depth it offers.

It has become very popular Java interview question in almost any senior or mid-senior level Java interviews. Investment banks mostly prefer to ask this question and sometimes even ask you to implement your own HashMap based upon your coding aptitude. The introduction of ConcurrentHashMap and other concurrent collections has also made this questions as starting point to delve into a more advanced feature. let's start the journey.

Double checking Singleton

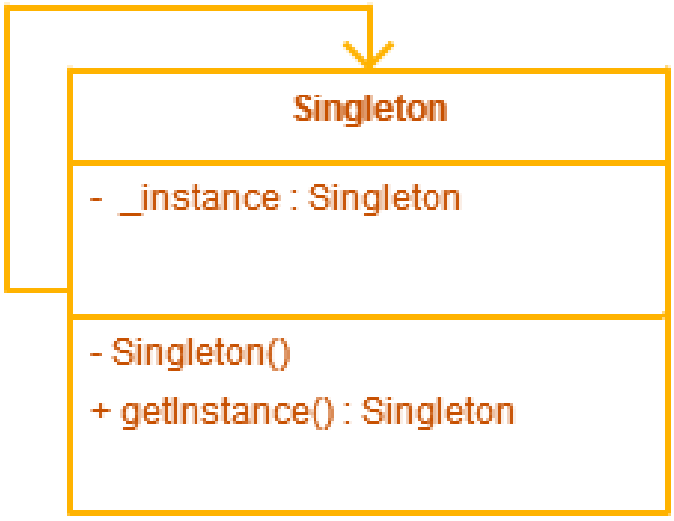
Singleton class is quite common among Java developers, but it poses many challenges to junior developers. One of the key challenge they face is how to keep Singleton class as Singleton? i.e. how to prevent multiple instances of a Singleton due to whatever reasons. *Double checked locking of Singleton* is a way to ensure only one instance of Singleton class is created through application life cycle. As name suggests, in double checked locking, code checks for an existing instance of Singleton class twice with and without locking to double ensure that no more than one instance of singleton gets created. By the way, it was broken before Java fixed its memory models issues in JDK 1.5. In this article, we will see *how to write code for double checked locking of Singleton in Java*, why double checked locking was broken before Java 5 and How that was fixed.

By the way this is also important from interview point of view, I have heard it's been asked to code double checked locking of Singleton by hand on companies in both financial and service sector, and believe me it's tricky, until you have clear understanding of what you are doing.

Why you need Double checked Locking of Singleton Class?

One of the common scenario, where a Singleton class breaks its contracts is multi-threading.

By the way this is not the best way to create thread-safe Singleton, you can use Enum as Singleton, which provides inbuilt `thread-safety` during instance creation. Another way is to use static holder pattern.



Without `volatile` modifier it's possible for another thread in Java to see half initialized state of _instance variable, but with `volatile` variable guaranteeing happens-before relationship, all the write will happen on volatile `_instance` before any read of `_instance` variable.

Factory Pattern

Factory design pattern in Java one of the core design pattern which is used heavily not only in JDK but also in various Open Source framework such as Spring, Struts and Apache along with decorator design pattern in Java. Factory Design pattern is based on **Encapsulation** object oriented concept. Factory method is used to create different object from factory often refereed as Item and it encapsulate the creation code. So instead of having object creation code on client side we encapsulate inside **Factory method in Java**. One of the best examples of factory pattern in Java is `BorderFactory` Class of Swing API.

What is static factory method or factory design pattern

Factory design pattern is used to create objects or Class in Java and it provides loose coupling and high cohesion. Factory pattern encapsulate object creation logic which makes it easy to change it later when you change how object gets created or you can even introduce new object with just change in one class. In GOF pattern list Factory pattern is listed as Creation design pattern. Factory should be an interface and clients first either creates factory or get factory which later used to create objects.

Example of static factory method in JDK

Best Example of Factory method design pattern is `valueOf()` method which is there in String and wrapper classes like Integer and Boolean and used for type conversion i.e. from converting String to Integer or String to double in java..

Some more examples of factory method design pattern from JDK is :

`valueOf()` method which returns object created by factory equivalent to value of parameter passed.

`getInstance()` method which creates instance of Singleton class.

`newInstance()` method which is used to create and return new instance from factory method every time called.

`getType()` and `newType()` equivalent of `getInstance()` and `newInstance()` factory method but used when factory method resides in separate class.

Problem which is solved by Factory method Pattern in Java

Whenever we talk about **object oriented language** it will based upon some concept like abstraction, polymorphism etc and on that encapsulation and delegation are important concept any design will be called good if task are delegated to different object and some kind of encapsulation is there.

Some time our application or framework will not know that what kind of object it has to create at run-time it knows only the interface or abstract class and as we know we can not create object of interface or abstract class so main problem is frame work knows **when** it has to create but don't know **what kind** of object.

Whenever we create object using `new()` we violate **principle of programming for interface rather than implementation** which eventually result in inflexible code and difficult to change in maintenance. By using Factory design pattern in Java we get rid of this problem.

Another problem we can face is class needs to contain objects of other classes or class hierarchies within it; this can be very easily achieved by just using the new keyword and the class constructor. The problem with this approach is that it is a very hard coded approach to create objects as this creates dependency between the two classes.

So **factory pattern** solve this problem very easily by model an interface for creating an object which at creation time can let its subclasses decide which class to instantiate, Factory Pattern promotes loose coupling by eliminating the need to bind application-specific classes into the code. The **factory methods** are typically implemented as virtual methods, so this pattern is also referred to as the "**Virtual Constructor**". These methods create the objects of the products or target classes.

When to use Factory design pattern in Java

- Static Factory methods are common in frameworks where library code needs to create objects of types which may be sub classed by applications using the framework.
- Some or all concrete products can be created in multiple ways, or we want to leave open the option that in the future there may be new ways to create the concrete product.
- Factory method is used when Products don't need to know how they are created.
- We can use factory pattern where we have to create an object of any one of sub-classes depending on the data provided

Advantage of Factory method Pattern in Java:

Factory pattern in Java is heavily used everywhere including JDK, open source library and other frameworks.In following are main advantages of using Factory pattern in Java:

1*)* *Factory method design pattern* decouples the calling class from the target class, which result in less coupled and highly cohesive code?

E.g.: JDBC is a good example for this pattern; application code doesn't need to know what database it will be used with, so it doesn't know what database-specific driver classes it should use. Instead, it uses factory methods to get Connections, Statements, and other objects to work with. Which gives you flexibility to change your back-end database without changing your DAO layer in case you are using ANSI SQL features and not coded on DBMS specific feature?

2) Factory pattern in Java enables the subclasses to provide extended version of an object, because creating an object inside factory is more flexible than creating an object directly in the client. Since client is working on interface level any time you can enhance the implementation and return from Factory.

3) Another benefit of using *Factory design pattern in Java* is that it encourages consistency in Code since every time object is created using Factory rather than using different constructor at different client side.

4) Code written using Factory design pattern in Java is also easy to debug and troubleshoot because you have a centralized method for object creation and every client is getting object from same place.

Some more advantages of factory method design pattern is:

1. **Static factory method** used in factory design pattern enforces use of Interface than implementation which itself a good practice. for example:

```
Map synchronizedMap = Collections.synchronizedMap(new HashMap());
```

2. Since static factory method have return type as Interface, it allows you to replace implementation with better performance version in newer release.

3. Another advantage of static factory method pattern is that they can cache frequently used object and eliminate duplicate object creation. `Boolean.valueOf()` method is good example which caches true and false boolean value.

4. Factory method pattern is also recommended by Joshua Bloch in Effective Java.

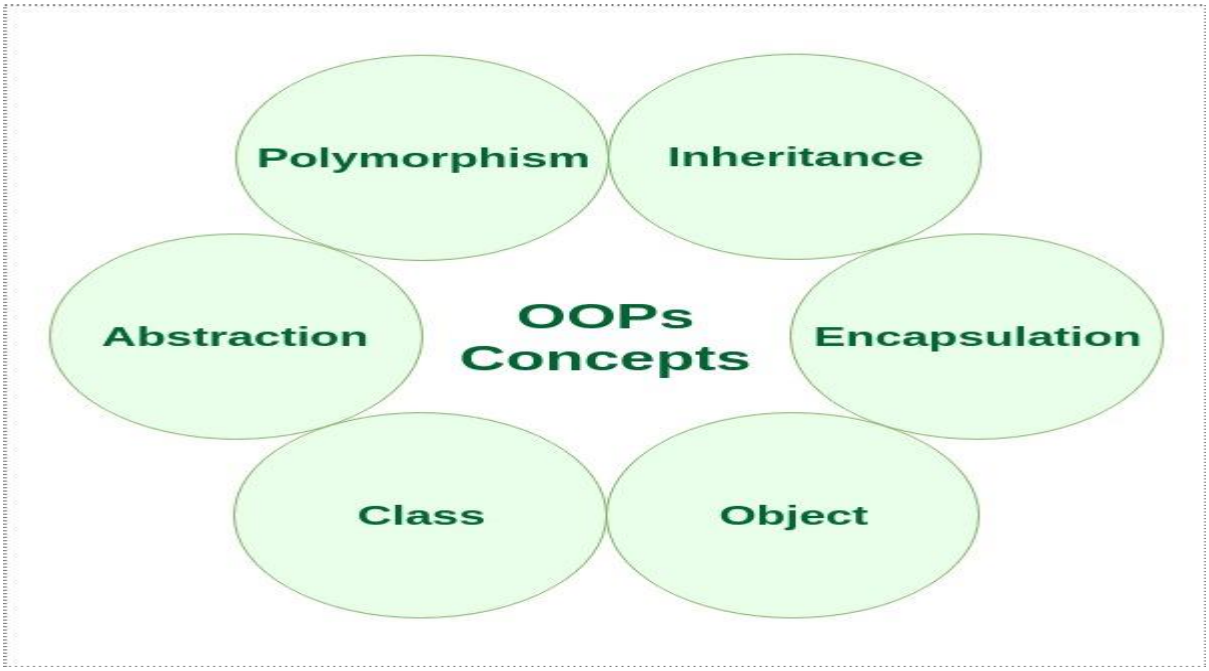
5 Factory method pattern offers alternative way of creating object.

6. Factory pattern can also be used to hide information related to creation of object.

OOPS

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Characteristics of an Object Oriented Programming language



Class: The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

- A Class is a user-defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behaviour of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage etc and member functions can apply brakes, increase speed etc.

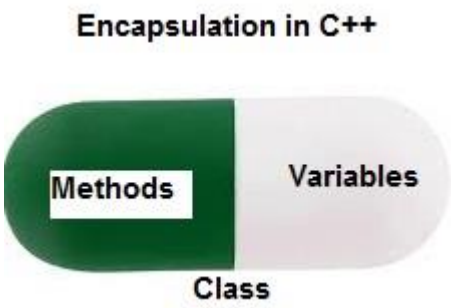
We can say that a **Class in C++** is a blue-print representing a group of objects which shares some common properties and behaviours.

Object: An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.
Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

Encapsulation: In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them. Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.



Encapsulation in C++

Encapsulation also leads to *data abstraction or hiding*. As using encapsulation also hides the data. In the above example, the data of any of the section like sales, finance or accounts are hidden from any other section.

Abstraction: Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

- *Abstraction using Classes*: We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- *Abstraction in Header files*: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file.

Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

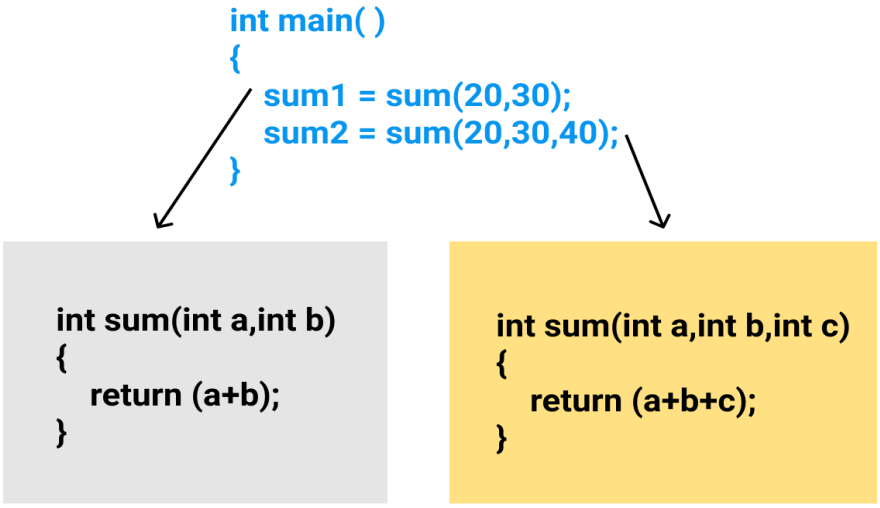
Polymorphism: The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

- *Operator Overloading*: The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
- *Function Overloading*: Function overloading is using a single function name to perform different types of tasks.
  Polymorphism is extensively used in implementing inheritance.

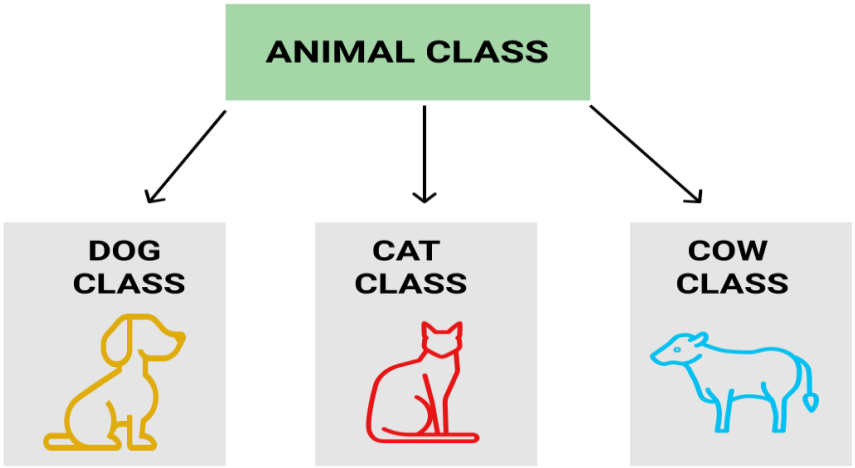
Example: Suppose we have to write a function to add some integers, some times there are 2 integers, some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.



Inheritance: The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class**:The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability**: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.



Dynamic Binding: In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Access Modifiers in C++

Access modifiers are used to implement an important feature of Object-Oriented Programming known as **Data Hiding**. Consider a real-life example:
The Indian secret informatic system having 10 senior members have some top secret regarding national security. So we can think that 10 people as class data members or member functions who can directly access secret information from each other but anyone can't access this information other than these 10 members i.e. outside people can't access information directly without having any privileges. This is what data hiding is.
Access Modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.
There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.

Let us now look at each one these access modifiers in details:

> **Public**: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

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

class Circle
{
    public:
        double radius;

        double compute_area()
        {
            return 3.14*radius*radius;
        }

};
  int main() {
    Circle obj;

    // accessing public datamember outside class
    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

Output:

```
Radius is: 5.5

Area is: 94.985
```

In the above program the data member *radius* is public so we are allowed to access it outside the class.

Private: The class members declared as *private* can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

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

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        double  compute_area()
        {   // member function can access private
            // data member radius
            return 3.14*radius*radius;
        }

};

int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class
    obj.radius = 1.5;

    cout << "Area is:" << obj.compute_area();
    return 0;
}
```

The output of above program will be a compile time error because we are not allowed to access the private data members of a class directly outside the class.

Output:

```
 In function 'int main()':
11:16: error: 'double Circle::radius' is private
         double radius;
                ^
31:9: error: within this context
     obj.radius = 1.5;
         ^
```

Protected: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

```
#include <bits/stdc++.h>
using namespace std;

// base class
class Parent
{
    // protected data members
    protected:
    int id_protected;

};

// sub class or derived class
class Child : public Parent
{


    public:
    void setId(int id)
    {

        // Child class is able to access the inherited
        // protected data members of base class

        id_protected = id;

    }

    void displayId()
    {
        cout << "id_protected is: " << id_protected << endl;
    }
};

// main function
```

```
int main() {

    Child obj1;

    // member function of the derived class can
    // access the protected data members of the base class

    obj1.setId(81);
    obj1.displayId();
    return 0;
}
```

MULTI-THREADING IN C++

Multithreading support was introduced in C+11. Prior to C++11, we had to use POSIX threads or p threads library in C. While this library did the job the lack of any standard language provided feature-set caused serious portability issues. C++ 11 did away with all that and gave us **std::thread**. The thread classes and related functions are defined in the **thread** header file.
std::thread is the thread class that represents a single thread in C++. To start a thread we simply need to create a new thread object and pass the executing code to be called (i.e, a callable object) into the constructor of the object. Once the object is created a new thread is launched which will execute the code specified in callable.
A callable can be either of the three

1) A function pointer
2) A function object
3) A lambda expression

After defining callable, pass it to the constructor.

```
import<thread>
std::thread thread_object(callable)
```

Launching thread using function pointer

The following code snippet demonstrates how this is done

```
void foo(param)
{
    // Do something
}

// The parameters to the function are put after the comma
std::thread thread_obj(foo, params);
```

Launching thread using lambda expression

The following code snippet demonstrates how this is done

```
// Define a lamda expression
auto f = [](params) {
    // Do Something
};

// Pass f and its parameters to thread
// object constructor as
std::thread thread_object(f, params);
```

We can also pass lambda functions directly to the constructor.

```
std::thread thread_object([](params) {
    // Do Something
};, params);
```

Waiting for threads to finish

Once a thread has started we may need to wait for the thread to finish before we can take some action. For instance, if we allocate the task of initializing the GUI of an application to a thread, we need to wait for the thread to finish to ensure that the GUI has loaded properly.

To wait for a thread use the **std::thread::join()** function. This function makes the current thread wait until the thread identified by ***this** has finished executing.
For instance, to block the main thread until thread t1 has finished we would do

A Complete C++ Program

A C++ program is given below. It launches three thread from the main function. Each thread is called using one of the callable objects specified above.

```
// CPP program to demonstrate multithreading
// using three different callables.
#include <iostream>
#include <thread>
using namespace std;

// A dummy function
void foo(int Z)
{
    for (int i = 0; i < Z; i++) {
        cout << "Thread using function"
               " pointer as callable\n";
    }
}

// A callable object
class thread_obj {
public:
    void operator()(int x)
    {
        for (int i = 0; i < x; i++)
            cout << "Thread using function"
                  " object as  callable\n";
    }
};

int main()
{
    cout << "Threads 1 and 2 and 3 "
```

```
         "operating independently" << endl;

    // This thread is launched by using
    // function pointer as callable
    thread th1(foo, 3);

    // This thread is launched by using
    // function object as callable
    thread th2(thread_obj(), 3);

    // Define a Lambda Expression
    auto f = [](int x) {
        for (int i = 0; i < x; i++)
            cout << "Thread using lambda"
             " expression as callable\n";
    };

    // This thread is launched by using
    // lamda expression as callable
    thread th3(f, 3);

    // Wait for the threads to finish
    // Wait for thread t1 to finish
    th1.join();

    // Wait for thread t2 to finish
    th2.join();

    // Wait for thread t3 to finish
    th3.join();

    return 0;
}
```

Output (Machine Dependent)

```
Threads 1 and 2 and 3 operating independently
Thread using function pointer as callable
Thread using lambda expression as callable
Thread using function pointer as callable
Thread using lambda expression as callable
Thread using function object as  callable
Thread using lambda expression as callable
Thread using function pointer as callable
Thread using function object as  callable
Thread using function object as  callable
```

Note:
To compile programs with std::thread support use

```
g++ -std=c++11 -pthread
```

Difference Between Thread Class and Runnable Interface in Java

A thread can be defined in two ways. First, by **extending a Thread class** that has already implemented a Runnable interface. Second, by directly **implementing a Runnable interface**. When you define a thread by extending Thread class you have to override the run() method in Thread class. When you define a thread implementing a Runnable interface you have to implement the only run() method of Runnable interface.

The basic difference between Thread and Runnable is that each thread defined by extending Thread class creates a unique object and get associated with that object. On the other hand, each thread defined by implementing Runnable interface shares the same object. Let us observe some other differences between Thread and Runnable with the help of comparison chart shown below:

Comparison Chart

BASIS FOR COMPARISON	THREAD	RUNNABLE
Basic	Each thread creates a unique object and gets associated with it.	Multiple threads share the same objects.
Memory	As each thread create a unique object, more memory required.	As multiple threads share the same object less memory is used.
Extending	In Java, multiple inheritance not allowed hence, after a	If a class define thread implementing the Runnable

BASIS FOR COMPARISON	THREAD	RUNNABLE
	class extends Thread class, it can not extend any other class.	interface it has a chance of extending one class.
Use	A user must extend thread class only if it wants to override the other methods in Thread class.	If you only want to specialize run method then implementing Runnable is a better option.
Coupling	Extending Thread class introduces tight coupling as the class contains code of Thread class and also the job assigned to the thread	Implementing Runnable interface introduces loose coupling as the code of Thread is separate form the job of Threads.

Definition of Thread Class

Thread is a class in **java.lang** package. The Thread class extends an **Object** class, and it implements **Runnable** interfaces. The Thread class has constructors and methods to create and operate on the thread. When we create multiple threads, each thread creates a unique object and get associated with that object. If you create a thread extending Thread class, further you can not extend any other class as java does not support multiple inheritance.

So, you should choose to extend Thread class only when you also want to override some other methods of Thread class.

Definition of Runnable Interface

Runnable is an interface in **java.lang** package. Implementing Runnable interface we can define a thread. Runnable interface has a single method **run(),** which is implemented by the class that implements Runnable interface. When you choose to define thread implementing a Runnable interface you still have a choice to extend any other class. When you create multiple threads by implementing Runnable interface, each thread shares the same runnable instance.

Key Differences Between Thread and Runnable in Java

1. Each thread created by extending the Thread class creates a unique object for it and get associated with that object. On the other hand, each thread created by implementing a Runnable interface share the same runnable instance.
2. As each thread is associated with a unique object when created by extending Thread class, more memory is required. On the other hand, each thread created by implementing Runnable interface shares same object space hence, it requires less memory.
3. If you extend the Thread class then further, you can inherit any other class as Java do not allow multiple inheritance whereas, implementing Runnable still provide a chance for a class to inherit any other class.
4. One must extend a Thread class only if it has to override or specialise some other methods of Thread class. You must implement a Runnable interface if you only want to specialise run method only.
5. Extending the Thread class introduces tight coupling in the code as the code of Thread and job of thread is contained by the same class. On the other hand, Implementing Runnable interface introduces loose coupling in the code as the code of Thread is seprate from the job assigned to the thread.

Conclusion

It is preferred to implement a Runnable interface instead of extending Thread class. As implementing Runnable makes your code loosely coupled as the code of thread is different from the class that assign job to the thread. It requires less memory and also allows a class to inherit any other class.

How HashSet Works Internally In Java?

HashSet uses HashMap internally to store it's objects. Whenever you create a HashSet object, one **HashMap** object associated with it is also created. This HashMap object is used to store the elements you enter in the HashSet. The elements you add into HashSet are stored as **keys** of this HashMap object. The value associated with those keys will be a **constant**.

Every constructor of HashSet class internally creates one HashMap object. You can check this in the source code of HashSet class in JDK installation directory. Below is the some sample code of the constructors of HashSet class.

Whenever you insert an element into HashSet using **add()** method, it actually creates an entry in the internally backing HashMap object with element you have specified as it's key and constant called "**PRESENT**" as it's value. This "PRESENT" is defined in the HashSet class as below.

```
1    // Dummy value to associate with an Object in the backing Map
2    private static final Object PRESENT = new Object();
```

Let's have a look at add() method of HashSet class.

```
1    public boolean add(E e)
2    {
3            return map.put(e, PRESENT)==null;
4    }
```

You can notice that, add() method of HashSet class internally calls put() method of backing HashMap object by passing the element you have specified as a key and constant "PRESENT" as it's value.

remove() method also works in the same manner.

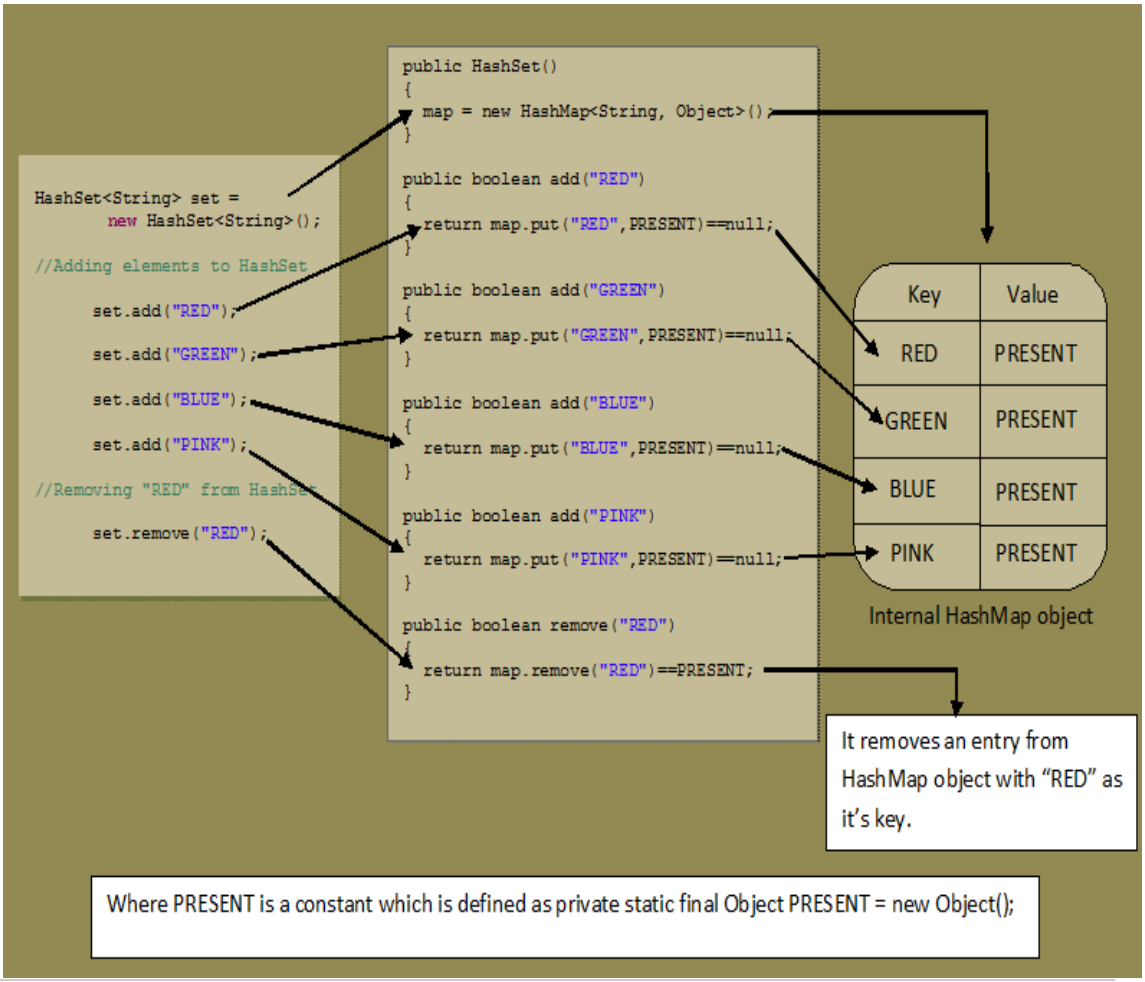
```
1    public boolean remove(Object o)
2    {
3            return map.remove(o)==PRESENT;
4    }
```

Let's see one example of HashSet and how it maintains HashMap internally.

```
1    public class HashSetExample
2    {
3        public static void main(String[] args)
4        {
5            //Creating One HashSet object
6
7            HashSet<String> set = new HashSet<String>();

             //Adding elements to HashSet
```

```
8
9               set.add("RED");
10
11              set.add("GREEN");
12
13              set.add("BLUE");
14
15              set.add("PINK");
16
17              //Removing "RED" from HashSet
18
19              set.remove("RED");
20          }
21      }
22
23
```

See the below picture how above program works internally. You can observe that internal HashMap object contains elements of HashSet as keys and constant "PRESENT" as their value.



Producer-Consumer solution

In computing, the producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem
To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Solution
The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Implementation of Producer Consumer Class

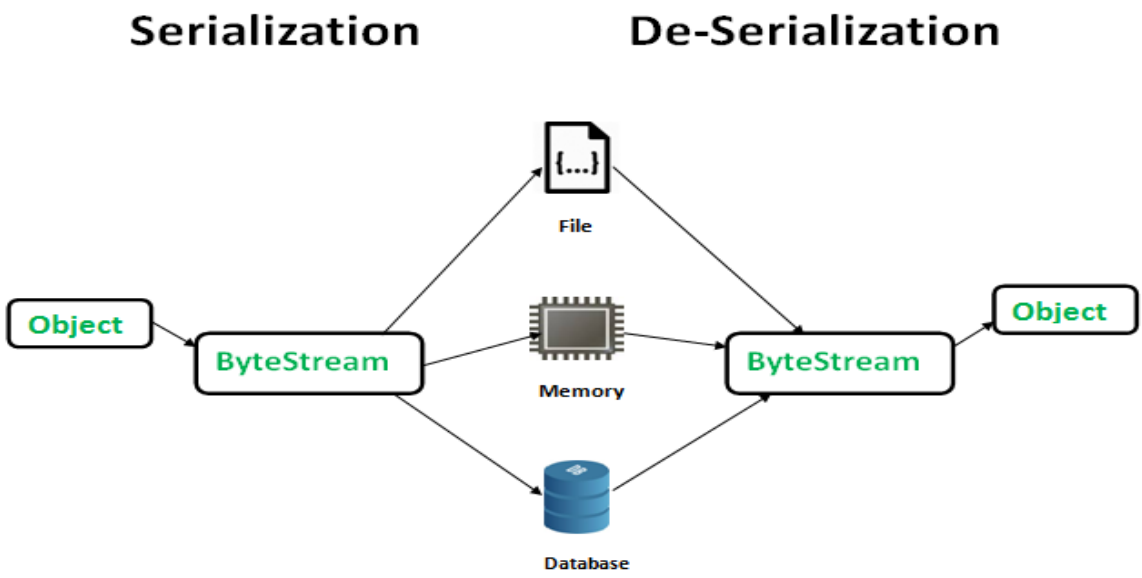
- A **LinkedList list** – to store list of jobs in queue.
- **A Variable Capacity** – to check for if the list is full or not
- A mechanism to control the insertion and extraction from this list so that we do not insert into list if it is full or remove from it if it is empty.

```
#include
<iostream>
            #include <pthread.h>
            #include <semaphore.h>
            #include <random>
            #include <unistd.h>


            using namespace std;


            #define BUFFER_SIZE 10


            int buffer[BUFFER_SIZE];
            int index=0;
```

```
sem_t full,empty;
pthread_mutex_t mutex;


void* produce(void* arg){
        while(1){
                sleep(1);
                sem_wait(&empty);
                pthread_mutex_lock(&mutex);
                int item = rand()%100;
                buffer[index++] = item;
                cout<<"Produced "<<item<<endl;
                pthread_mutex_unlock(&mutex);
                sem_post(&full);
        }
}


void* consume(void* arg){
        while(1){
                sleep(1);
                sem_wait(&full);
                pthread_mutex_lock(&mutex);
                int item = buffer[--index];
                cout<<"Consumed "<<item<<endl;
                pthread_mutex_unlock(&mutex);
                sem_post(&empty);
        }
}


int main(){
        pthread_t producer,consumer;
        sem_init(&empty,0,BUFFER_SIZE);
        sem_init(&full,0,0);
        pthread_mutex_init(&mutex,NULL);
        pthread_create(&producer,NULL,produce,NULL);
        pthread_create(&consumer,NULL,consume,NULL);
        pthread_exit(NULL);
}
```

Serialization and Deserialization

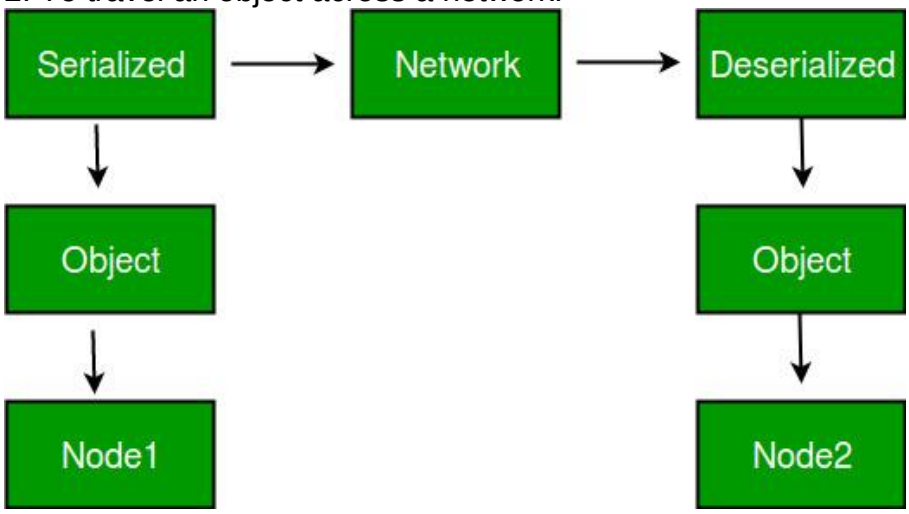
Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.



The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

Advantages of Serialization

1. To save/persist state of an object.
2. To travel an object across a network.



TreeSet comparator()

The java.util.TreeSet.comparator() method shares an important function of setting and returning the comparator that can be used to order the elements in a TreeSet. The method returns Null value if the set follows the natural ordering pattern of the elements.

Syntax:

```
comp_set = (TreeSet)tree_set.comparator()
```

Parameters: The method does not take any parameters.

Return Value: The method returns the comparator set used to order the elements of the set in a specific order. It returns a Null value if the set follows the default or natural ordering pattern.

END