1. ***Thread***
2. ***Singleton Pattern***
3. ***Strategy Pattern***

***Process:***

Process has a self-contained execution environment. Each process has its own memory space.

***Thread:***

Light weight processes.

Creating a thread requires fewer resources than process.

Thread exists within a process. Every process has a at least one thread.

Thread’s share the process’s resources, including memory and open files.

https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html

After creating instance of thread we have to write code to run it.

There are two ways to run the thread

1.Provide a Runnable Object

**public class HelloRunnable implements Runnable {**

**public void run() {**

**System.out.println("Hello from a thread!");**

**}**

**public static void main(String args[]) {**

**(new Thread(new HelloRunnable())).start();**

**}**

**}**

2.Subclass thread

**public class HelloThread extends Thread {**

**public void run() {**

**System.out.println("Hello from a thread!");**

**}**

**public static void main(String args[]) {**

**(new HelloThread()).start();**

**}**

**}**

*Sleep*

Thread.sleep causes the current thread to suspend execution for specific period.

This is an efficient way of making the processor time available to threads in the application or to other applications.

Two overloaded versions of sleep are provided.

One with milliseconds

One with nanoseconds

Sleep period can be terminated by interrupts.

*Joins*

The join method allows the one thread to wait for the completion of another.

If a t is thread object whose thread is currently executing t.join() causes current thread to pause execution

until t’s thread terminates.

Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.

Like sleep, join responds to an interrupt by exiting with an InterruptedException

Interrupts

An interrupt is an indication to the thread that it should stop what it is doing and do something else.

A thread sends interrupt by invoking **interrupt** on thread object for the thread to be interrupted. For the interrupt mechanism

to work properly, the interrupted thread must support its own interruption.

Using Thread.interrupted() method, we can check whether thread is interrupted or not.

When a thread checks for an interrupt by invoking the static method Thread.interrupted, interrupt status is cleared. The non-

static isInterrupted method, which is used by one thread to query the interrupt status of another, does not change the

interrupt status flag.

Synchronization

Threads communicate directly by sharing access to fields and the objects reference fields refer to.

There might be 2 errors possible

1.Thread interference

2.Memory Consistency

Synchronization can introduce Thread Contention.

Thread Contention

It occurs when two or more threads try to access the same resource simultaneously and cause the java runtime to execute one

Or more threads slowly or even suspend their execution

Thread interference

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

Memory Consistency

*Memory consistency errors* occur when different threads have inconsistent views of what should be the same data.

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object

is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods. (An

important exception: final fields, which cannot be modified after the object is constructed, can be safely read through non-

synchronized methods, once the object is constructed) This strategy is effective, but can present problems with [liveness](https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html),

<https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>

Constructors cannot be synchronized. Using a synchronized keyword with a constructor is a syntax error.

<https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html>

Executors allow us to separate thread creation and management from rest of the application

<https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html>

Executor Interfaces

1.Executor

2.Executor Service

3.Scheduled Executor Service

*Constructors*

Constructors are never inherited.

Constructor will be called when we instantiate any new object

The purpose of java constructor is to initialize the newly created object before it is used.

Constructor will have no return type.

Sometimes there is a need of initializing an object in different ways. This can be done using constructor

overloading. Different constructors can do different work by implementing different line of codes and

are called based on the type and no of parameters passed.

According to the situation , a constructor is called with specific number of parameters among overloaded constructors.

Constructors in the super class can be called using super() method.

You can’t extend a class with a private constructor.

*Singleton Pattern*

It ensures a class has only one instance and provides global point of access to it.

/\*

\* 1st version: creates multiple instance if two thread access

\* this method simultaneously

\*/

private static Singleton uniqueInstance;

private Singleton() {}

public static Singleton getInstance() {

if (uniqueInstance == null) {

uniqueInstance = new Singleton();

}

return uniqueInstance;

}

/\*

\* 2nd version : this definitely thread-safe and only

\* creates one instance of Singleton on concurrent environment

\* but unnecessarily expensive due to cost of synchronization

\* at every call.

\*/

public static synchronized Singleton getInstanceTS() {

if (\_instance == null) {

\_instance = new Singleton();

}

return \_instance;

}

/\*

\* 3rd version : An implementation of double checked locking of Singleton.

\* Intention is to minimize cost of synchronization and improve performance,

\* by only locking critical section of code, the code which creates

instance of Singleton class.

\* By the way this is still broken, if we don't make \_instance volatile,

as another thread can

\* see a half initialized instance of Singleton.

\*/

public class Singleton {

private volatile static Singleton uniqueInstance;

private Singleton() {}

public static Singleton getInstance() {

if (uniqueInstance == null) {

synchronized (Singleton.class) {

if (uniqueInstance == null) {

uniqueInstance = new Singleton();

}

}

}

return uniqueInstance;

}  
The Singleton Pattern ensures

you have at most one instance

of a class in your application.

The Singleton Pattern also

provides a global access point

to that instance.

Java’s implementation of the

Singleton Pattern makes use

of a private constructor, a static

method combined with a static

variable.

Examine your performance

and resource constraints and

carefully choose an appropriate

Singleton implementation for

multithreaded applications

(and we should consider all

applications multithreaded!).

Beware of the double-checked

locking implementation; it is not

thread-safe in versions before

Java 2, version 5.

Be careful if you are using

multiple class loaders; this

could defeat the Singleton

implementation and result in

multiple instances.

**Adapter Pattern**

Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

Blocking Queue:

It is a FIRST IN FIRST OUT data structure which blocks or times out when we try to add an element

To queue or try to retrieve value from an empty queue.

Factory pattern:

Observer Pattern:

It defines one to many dependency between objects so that when one object changes the state , all of its dependents are notified and updated automatically.

Sorting Algorithms

Linear Time:

An algorithm is said to take **linear time**, or *O*(*n*) time, if its time complexity is *O*(*n*). Informally, this means that the running time increases at most linearly with the size of the input. More precisely, this means that there is a constant *c* such that the running time is at most *cn* for every input of size *n*. For example, a procedure that adds up all elements of a list requires time proportional to the length of the list, if the adding time is constant, or, at least, bounded by a constant.

Comparison Sorts

1.Bubble Sort

2.Merge Sort

Bubble Sort

Bubble Sort is an elementary sorting algorithm. It works by repeatedly exchanging adjacent elements, if necessary.

public static void sort(int[] input) {

int inputLength = input.length;

int temp;

boolean is\_sorted;

for (int i = 0; i < inputLength; i++) {

is\_sorted = true;

for (int j = 1; j < (inputLength - i); j++) {

if (input[j - 1] > input[j]) {

temp = input[j - 1];

input[j - 1] = input[j];

input[j] = temp;

is\_sorted = false;

}

}

// is sorted? then break it, avoid useless loop.

if (is\_sorted) break;

System.out.println("\n");

}

}

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.

// Java program for implementation of Selection Sort

class SelectionSort

{

void sort(int arr[])

{

int n = arr.length;

// One by one move boundary of unsorted subarray

for (int i = 0; i < n-1; i++)

{

// Find the minimum element in unsorted array

int min\_idx = i;

for (int j = i+1; j < n; j++)

if (arr[j] < arr[min\_idx])

min\_idx = j;

// Swap the found minimum element with the first

// element

int temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

}

// Prints the array

void printArray(int arr[])

{

int n = arr.length;

for (int i=0; i<n; ++i)

System.out.print(arr[i]+" ");

System.out.println();

}

// Driver code to test above

public static void main(String args[])

{

SelectionSort ob = new SelectionSort();

int arr[] = {64,25,12,22,11};

ob.sort(arr);

System.out.println("Sorted array");

ob.printArray(arr);

}

}

Decorator Pattern