Main Goles to achieve in project: --

Safe piece extensibility has been an ideal property of working frameworks, and has been an objective of critical exploration. Extensibility is broadly utilized however expansion security has slacked behind.

Expansion security is particularly significant today due to the broad utilization of compartments in the cloud. Existing arrangements are heavyweight and add huge improvement cost or on the other hand runtime overhead.

We dissect the present status of Linux augmentation blunders what's more, investigate the achievability of utilizing a significant level, type-safe language to permit designers to compose Linux bit expansions in Rust.

We distinguish key difficulties to this methodology what's more, propose potential arrangements. In view of our experience, this approach can possibly give appealing wellbeing and execution advantages to Linux expansions.

In advanced object-oriented programming languages like java, .NET we can reduce the time and improve the reusability and extensibility.

Memory management:--

I would like to mention the memory management is a process in the .net environment.

1)stack and heap

2)value type and reference type

The CLR allocated the memory for value type and reference type variables.

Stack:-

Stack is straightforward Last in the first out structure stack is extremely quick for assigning memory.

At the time program aggregation memory will appoint to both worth sort and reference types.

At the point when the strategy arranges then, CLR will reset the past bookmarks memory will assigning in basic activity.

Heap:-

The load can be seen as an irregular tangle of items. It permits objects to be designated or deallocated in an arbitrary order.

variables distributed on the pile have their memory dispensed at run time and getting to the memory is a bit slower however, pile size is just restricted by the size of virtual memory. Value type factors are put away in the stack memory and reference type worth will store in pile memory.

Value type variables data will be within its memory location.

Value types --> struct,enum,ushort,ulong,uint,long,int,float,double,decimal,char,bool,byte.

Reference type will hold the real data in run time.

Refrence types -->arrays,dynamic,object,string,delegate,interface,class.

Example for value type variables :-

static void Main(){

int a=2;

int b=a;

a=8;

console.WriteLine("a=" + a + "");

console.WriteLine("b=" + b + "");

}

output:--

a= 8

b= 2

Both worth sort factors will work independently. If we allow the reference type variable to another variable then the reference type variable will address the location of the variable, the reference is duplicated and the two factors highlight a similar area of the stack memory.

Example for reference variables :--

class Order

{

public Int64 OrderId { get; set; }

public string RefrenceId { get; set; }

}

static void Main(string[] args)

{

Order aliceobjOrder = new Order() { OrderId = 1,RefrenceId="Ord\_001"};

Order objOrder = aliceobjOrder;

Console.WriteLine("Order refrence id is"+ aliceobjOrder.RefrenceId + "objOrder refrence id is" + objOrder.RefrenceId +"");

}

The memory space for a variable is put away on either the stack or the store. It relies upon the setting where it is pronounced. Every nearby factor (for example one announced in a strategy) is put away on the stack. That incorporates reference type factors — the actual variable is on the stack, yet recall that the worth of a reference type variable is just a reference (or invalid), not simply the item. Strategy boundaries consider nearby factors as well, yet in case they are proclaimed with the ref modifier, they don't get their own space, yet share an opening with the variable utilized in the calling code. Case factors for a reference type are consistently on the pile. That is the place where the actual article lives. Instance factors for a worth sort are put away in the very setting as the variable that proclaims the worth kind. The memory space for the occasion adequately holds the openings for each field inside the occurrence. That implies (given the past two focuses) that a struct variable announced inside a strategy will consistently be on the stack, while a struct variable which is an example field of a class will be on the store. Each static variable is put away on the load, whether or not it's pronounced inside a reference type or a worth sort. There is just one space altogether regardless of the number of occasions are made.

The second is Wallace Kelly's YouTube video. I unequivocally suggest you watch this video as he exhibits the subject impeccably. After watching that, you will perceive how everything becomes alright to you.