Internship Program

Backend (.NET)

Internship In EFG Hermes>>LinkedIn Profile

Under supervision

Manager Engineering In Backend

Eng / Mohamed Bayoumi>>LinkedIn Profile

This Topics Through Internship 1'ST Day (4/8/2024)

1-Low Language, High Language , Assembly.

2-OOP Language

3-Serialization / Deserialization

Prepared By The Trainee

Ali Wazeer >>LinkedIn Profile

GitHub>>>EFG_Hermes-Internship

Jul/2024

NOte *“You must see the Readme.md Files files because they contain an explanation of each topic ”*

1-High Level Language Vs. Low Level Language Vs Assembly Language.

-High Level Language
High-level languages are programming languages that are used for writing programs or software that can be understood by humans and computers. High-level languages are easier to understand for humans because they use a lot of symbols, letters, phrases to represent logic and instructions in a program. It contains a high level of abstraction compared to low-level languages.

-Low Level Language
low level languages which are closer to hardware as compared to high-level languages instead of software. They provide little or no abstraction from the machine instructions and that’s why they allow programmers to manipulate hardware elements like register, memory etc. Low-level languages are often used for designing systems, such as developing operating systems, device drivers, and embedded systems.

-Assembly Language
Assembly language is a low-level language that helps to communicate directly with computer hardware. It uses mnemonics to represent the operations that a processor has to do. Which is an intermediate language between high-level languages like C++ and the binary language. It uses hexadecimal and binary values, and it is readable by humans.

Machine-level language The machine-level language is a language that consists of a set of instructions that are in the binary form 0 or 1. As we know that computers can understand only machine instructions, which are in binary digits, i.e., 0 and 1, so the instructions given to the computer can be only in binary codes. Creating a program in a machine-level language is a very difficult task as it is not easy for the programmers to write the program in machine instructions. It is error-prone as it is not easy to understand, and its maintenance is also very high. A machine-level language is not portable as each computer has its machine instructions, so if we write a program in one computer it will no longer be valid in another computer.

Characteristics Of Low Level Language….

-It can be understood easily by the machine.

-It is considered as a machine-friendly language.

-It is difficult to understand.

-It is difficult to debug.

-Its maintenance is also complex.

-It is not portable.

-It depends on the machine; hence it can't be run on different platforms.

-It requires an assembler that would translate instructions.

-It is not used widely in today's times.

Characteristics of high level programming languages

1) Easy to Learn
2) Easy Error Detection
3) Standardized Syntax
4) Deep Hardware Knowledge not Required
5) Machine Independence
6) More Programmers
7) Shorter Programs

Characteristics of Assembly level programming languages

- Assembly language uses mnemonic instructions to represent machine code instructions.
- Direct access to hardware.
- Low-level abstraction.
- Efficient use of resources.
- Full control over program flow.

- Direct access to memory.
- Better control over CPU.

Assembly Instructions

Here are some common assembly language instructions and their descriptions:

1. **Data Transfer Instructions**:

   - `mov` - Move data between registers or memory

   - `push` - Push a value onto the stack

   - `pop` - Pop a value from the stack

2. **Arithmetic and Logical Instructions**:

   - `add`, `sub`, `mul`, `div` - Perform addition, subtraction, multiplication, and division

   - `and`, `or`, `xor`, `not` - Perform bitwise logical operations

3. **Control Transfer Instructions**:

   - `jmp` - Unconditional jump to a specified address

   - `je`, `jne`, `jz`, `jnz` - Conditional jumps based on flags (zero, carry, overflow, etc.)

   - `call`, `ret` - Call a subroutine and return from it

4. **Comparison Instructions**:

   - `cmp` - Compare two operands and set flags accordingly

5. **String Instructions**:

   - `mov`, `cmps`, `scas`, `lods`, `stos` - Perform operations on string data

6. **Miscellaneous Instructions**:

   - `nop` - No operation (used for padding)

   - `int` - Trigger a software interrupt

   - `syscall` - Make a system call (on Linux)

   - `hlt` - Halt the processor

Object-Oriented Programming (OOP)

Object-oriented programming (OOP) is a programming paradigm that is based on the concept of “objects”, which are instances of a class. These objects contain both data (attributes) and behavior (methods) that describe their characteristics and actions.

Functional Programming (FP)

Functional programming (FP) is a programming paradigm that is based on the concept of “functions” that take inputs and produce outputs. These functions are pure, meaning they don’t have side-effects, and are first-class citizens, meaning they can be passed around as arguments or returned as results.

Object-Oriented Programming OOP is based on the following four principles:

Encapsulation

Encapsulation allows you to hide the internal state and behavior of an object from the outside world. At the same time it allows the object to be accessed only through a well-defined interface. This means that the internal state of an object is not directly accessible from outside the object, and can only be modified or accessed through the object’s methods. It also allows for the implementation of an object to change without affecting the code that uses the object. This can make the code more robust and maintainable over time.

1. **Classes**: A class is the basic building block of encapsulation in C#. A class defines the data (fields) and the operations (methods) that the object can perform.

2. **Access Modifiers**: C# provides four access modifiers that you can use to control the visibility and accessibility of class members (fields, properties, methods, and events):

- `public`: The member is accessible from anywhere.

- `private`: The member is accessible only within the same class.

- `protected`: The member is accessible within the same class and its derived classes.

- `internal`: The member is accessible within the same assembly (project).

```
class BankAccount {

  constructor(balance) {

    this._balance = balance;

  }

  deposit(amount) {

    this._balance += amount;

  }

  withdraw(amount) {

    if (this._balance >= amount) {

      this._balance -= amount;

    } else {

      console.log("Insufficient funds");

    }

  }

  get balance() {

    return this._balance;

  }
}

const account = new BankAccount(1000);

console.log(account.balance); // 1000

account.deposit(500);

console.log(account.balance); // 1500

account.withdraw(300);

console.log(account.balance); // 1200
```

In this example, the `BankAccount` class has a private field `_balance` that represents the account balance. The class provides three public methods: `Deposit`, `Withdraw`, and `GetBalance`. These methods are the public interface of the `BankAccount` class, and they allow the outside world to interact with the account without directly accessing the `_balance` field.

Inheritance

is another fundamental concept in object-oriented programming (OOP) that allows you to create new classes based on existing ones. In C#, inheritance is achieved through the use of the `class` keyword and the `:` operator.

1. **Base Class and Derived Class**: The existing class is called the **base class**, and the new class that inherits from the base class is called the **derived class**.

2. **Inheritance Syntax**: To create a derived class, you use the `:` operator followed by the name of the base class:

3. **Inheritance of Members**: When a class inherits from a base class, it automatically inherits all the public and protected members (fields, properties, methods, and events) of the base class. The derived class can then use these inherited members as if they were defined in the derived class itself.

4. **Access Modifiers**: The access modifiers (`public`, `protected`, `internal`, `private`) used in the base class determine the visibility and accessibility of the inherited members in the derived class.

5. **Overriding and Virtual Methods**: Derived classes can override the implementation of virtual methods defined in the base class by using the `virtual` and `override` keywords.

```
public class Animal

{

    public string Name { get; set; }

    public int Age { get; set; }

    public virtual void MakeSound()

    {

        Console.WriteLine("The animal makes a sound.");

    }

}

public class Dog : Animal

{

    public string Breed { get; set; }

    public override void MakeSound()

    {

        Console.WriteLine("The dog barks.");
    }

}

public class Cat : Animal

{

    public bool IsFriendly { get; set; }

    public override void MakeSound()

    {

        Console.WriteLine("The cat meows.");

    }

}
```

In this example, the `Animal` class is the base class, and `Dog` and `Cat` are the derived classes. The `Dog` and `Cat` classes inherit the `Name` and `Age` properties from the `Animal` class, and they also override the `MakeSound` method.

Inheritance allows you to create hierarchies of related classes, where the derived classes inherit the common characteristics and behaviors from the base classes. This promotes code reuse, simplifies development, and helps to create a more modular and maintainable codebase.

Polymorphism

- Lets objects of different classes to be treated as objects of a common base class, and to be used interchangeably.
- Allows for more flexibility and can make code more reusable.

- Enables the use of a common interface for different classes, making it possible to write code that can work with objects of different types without knowing their specific class.

```
Animal[] animals = { new Dog(), new Cat() };

foreach (var animal in animals)

{

    animal.MakeSound();

}

// Using interfaces

IAnimal[] animalList = { new Dog(), new Cat() };

foreach (var animal in animalList)

{

    animal.MakeSound();

}
```

Abstraction

- Abstraction is the ability to focus on the essential features of an object, and to ignore non-essential details.
- Allows for the creation of classes that are not tied to specific implementations, making the code more flexible and easy to maintain.

- Makes it possible to work with objects of a class without knowing the details of their implementation, which can make the code more robust and less error-prone.

```
// Base class

public class Animal

{

    public virtual void MakeSound()

    {

        Console.WriteLine("The animal makes a sound.");

    }

}

// Derived class

public class Dog : Animal

{

    public override void MakeSound()

    {

        Console.WriteLine("The dog barks.");

    }

}

// Derived class

public class Cat : Animal
```

```
{
    public override void MakeSound()
    {
        Console.WriteLine("The cat meows.");
    }
}
class Program
{
    static void Main(string[] args)
    {
        // Create instances of different animal types
        Animal animal1 = new Animal();
        Dog dog = new Dog();
        Cat cat = new Cat();

        // Call the MakeSound method on each object
        animal1.MakeSound(); // Output: The animal makes a sound.
        dog.MakeSound(); // Output: The dog barks.
        cat.MakeSound(); // Output: The cat meows.
        // Store different animal types in an array

        Animal[] animals = { new Animal(), new Dog(), new Cat() };


        // Call the MakeSound method on each animal in the array

        foreach (Animal a in animals)

        {

            a.MakeSound();

        }

    }

}
```

Serialization And Deserialization

Understanding XML Serialization and Deserialization

XML serialization is the process of converting objects into XML format, making it easier to store, transmit, or share data. Deserialization, on the other hand, involves converting XML data back into objects. This allows for seamless integration of structured data in C# applications.