C#

TEORIA DI BASE

C# (pronunciato "C Sharp") è un linguaggio di programmazione orientato agli oggetti sviluppato da Microsoft. È uno dei linguaggi di programmazione più utilizzati al mondo e viene spesso utilizzato per lo sviluppo di applicazioni desktop, applicazioni Web e giochi.

Ecco una breve panoramica su come programmare in C#:

1. Installare un ambiente di sviluppo integrato (IDE) come Visual Studio. Questo IDE fornisce un'interfaccia grafica utile per scrivere, compilare, eseguire e debuggare codice C#.
2. Creare un nuovo progetto e scegliere il tipo di applicazione da sviluppare, ad esempio un'applicazione desktop o una Web application.
3. Scrivere il codice C# nel file del progetto, usando un editor di codice integrato nell'IDE. C# è un linguaggio di programmazione orientato agli oggetti, quindi il

codice sarà organizzato in classi, metodi e proprietà.

4. Compilare il codice. L'IDE compila il codice sorgente in un file eseguibile o in una libreria di classi.
5. Eseguire il programma. L'IDE può eseguire il programma in modalità di debug o in modalità release.
6. Debuggare il programma. Se si verificano errori durante l'esecuzione del programma, l'IDE fornisce strumenti di debug per individuare e risolvere i problemi.

Ecco un esempio di codice C# che mostra come stampare un messaggio sulla console:

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Ciao, mondo!");
    }
}
```

In questo esempio, il codice utilizza il namespace `System` per accedere alla classe `Console`. La classe `Console` ha un metodo `WriteLine` che consente di scrivere un messaggio sulla console.

Ecco un esempio di codice C# che utilizza variabili per eseguire un calcolo e visualizzare il risultato sulla console:

```
using System;

class Program
{
    static void Main()
    {
        // dichiarazione delle variabili
        int numero1, numero2, risultato;

        // richiesta dell'input all'utente
        Console.WriteLine("Inserisci il primo numero:");
        numero1 = Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("Inserisci il secondo numero:");
        numero2 = Convert.ToInt32(Console.ReadLine());
```

```
        // calcolo del risultato
        risultato = numero1 + numero2;

        // visualizzazione del risultato sulla console
        Console.WriteLine("Il risultato della somma di {0} e {1} è {2}", numero1, nume
ro2, risultato);

        // attesa dell'input da parte dell'utente per chiudere la console
        Console.WriteLine("Premi un tasto per uscire...");
        Console.ReadKey();
    }
}
```

In questo esempio, abbiamo dichiarato tre variabili: `numero1`, `numero2` e `risultato`. Abbiamo poi richiesto all'utente di inserire due numeri utilizzando la classe `Console` e il metodo `ReadLine()`, che restituisce una stringa. Abbiamo quindi utilizzato il metodo `Convert.ToInt32()` per convertire le stringhe inserite dall'utente in numeri interi e assegnarli alle variabili `numero1` e `numero2`.

Successivamente, abbiamo eseguito il calcolo della somma dei due numeri inseriti dall'utente e assegnato il risultato alla variabile `risultato`. Infine, abbiamo utilizzato il metodo `Console.WriteLine()` per visualizzare il risultato sulla console.

Infine, abbiamo utilizzato il metodo `Console.ReadKey()` per attendere l'input da parte dell'utente prima di chiudere la console.

ARRAY

In C#, gli array sono un tipo di dato che consente di memorizzare una serie di elementi dello stesso tipo, raggruppati in un unico blocco di memoria. Gli elementi all'interno dell'array sono numerati e accessibili tramite un indice, che può essere utilizzato per leggere o scrivere il valore dell'elemento corrispondente.

Per dichiarare un array in C#, si utilizza la seguente sintassi:

```
tipo[] nomeArray = new tipo[dimensione];
```

dove `tipo` rappresenta il tipo di dato degli elementi dell'array, `nomeArray` è il nome scelto per l'array e `dimensione` è il numero di elementi che l'array deve contenere.

Ad esempio, per dichiarare un array di interi di dimensione 5, si può utilizzare il seguente codice:

```
int[] numeri = new int[5];
```

Per accedere ad un elemento dell'array, si utilizza la sintassi `nomeArray[indice]`, dove `indice` rappresenta la posizione dell'elemento all'interno dell'array. Ad esempio, per assegnare il valore 10 al primo elemento dell'array `numeri`, si può utilizzare il seguente codice:

```
numeri[0] = 10;
```

È anche possibile inizializzare un array durante la sua dichiarazione, specificando i valori degli elementi tra parentesi graffe, separati da virgola. Ad esempio, per creare un array di interi con i valori 1, 2, 3, 4 e 5, si può utilizzare il seguente codice:

```
int[] numeri = new int[] {1, 2, 3, 4, 5};
```

Gli array possono anche essere multidimensionali, ossia composti da più di una dimensione. Ad esempio, un array bidimensionale può essere dichiarato nel seguente modo:

```
tipo[,] nomeArray = new tipo[dimensione1, dimensione2];
```

dove `dimensione1` e `dimensione2` rappresentano le dimensioni dell'array.

Per accedere ad un elemento di un array multidimensionale, si utilizza la sintassi `nomeArray[indice1, indice2]`.

In C# esiste anche un tipo di dato chiamato `List`, che è simile ad un array ma consente di aggiungere e rimuovere elementi in modo dinamico, senza dover specificare la dimensione dell'array in anticipo. Le `List` sono implementate come array ridimensionabili internamente, quindi possono essere utilizzate in modo molto simile agli array.

LISTE

In C#, le `List` sono una struttura dati che rappresenta una sequenza di elementi ordinati. Sono molto utili per gestire insiemi di dati dinamici che possono espandersi o contrarsi durante l'esecuzione del programma.

Ecco un esempio di come utilizzare le `List` in C#:

```
using System;
using System.Collections.Generic;

namespace ListExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Creazione di una lista di stringhe
            List<string> shoppingList = new List<string>();

            // Aggiunta di elementi alla lista
            shoppingList.Add("Milk");
            shoppingList.Add("Bread");
            shoppingList.Add("Eggs");
            shoppingList.Add("Cheese");

            // Stampa del numero di elementi nella lista
            Console.WriteLine("Number of items in shopping list: " + shoppingList.Coun
t);

            // Stampa di tutti gli elementi della lista
            Console.WriteLine("Shopping list items:");
            foreach (string item in shoppingList)
            {
                Console.WriteLine(item);
            }

            // Rimozione di un elemento dalla lista
            shoppingList.Remove("Bread");

            // Stampa del numero di elementi nella lista dopo la rimozione
            Console.WriteLine("Number of items in shopping list after removal: " + sho
ppingList.Count);
        }
    }
}
```

In questo esempio, viene creata una lista di stringhe `shoppingList` e vengono aggiunti alcuni elementi alla lista. Viene poi stampato il numero di elementi nella lista utilizzando la proprietà `Count`, seguito dall'elenco di tutti gli elementi utilizzando un ciclo `foreach`. Infine, viene rimosso un elemento dalla lista utilizzando il metodo `Remove` e viene nuovamente stampato il numero di elementi nella lista.

Le `List` sono disponibili anche per altri tipi di dati, come `int`, `double`, `bool`, ecc. Ecco un esempio di come creare una lista di interi:

```
List<int> numbers = new List<int>();
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);
```

In questo esempio, viene creata una lista di interi `numbers` e vengono aggiunti alcuni elementi alla lista utilizzando il metodo `Add`.

CLASSI E OGGETTI

In C#, una **classe** rappresenta un modello o un prototipo per la creazione di oggetti. In altre parole, una classe è un insieme di dati e metodi che descrivono un tipo di oggetto.

Ad esempio, se si vuole creare un'applicazione che gestisce i dati degli utenti, si può definire una classe "Utente" che contiene i dati degli utenti come nome, cognome, indirizzo email, password, ecc. e i metodi associati come la registrazione, il login, la modifica dei dati dell'account, ecc.

Quando una classe viene istanziata, si crea un **oggetto** che corrisponde alla classe e che contiene le sue proprietà e i suoi metodi. Ad esempio, se si istanzia la classe "Utente", si crea un oggetto "Utente" con i dati e i metodi associati.

Ecco un esempio di una classe "Utente" in C#:

```
public class Utente {
   // proprietà
   public string Nome { get; set; }
   public string Cognome { get; set; }
   public string Email { get; set; }
   public string Password { get; set; }

   // metodi
   public void Registra() {
      // codice per la registrazione dell'utente
   }

   public void Accedi() {
      // codice per il login dell'utente
   }

   public void ModificaDati() {
      // codice per la modifica dei dati dell'utente
   }
}
```

Per creare un **oggetto** "Utente" a partire dalla classe sopra definita, si può utilizzare il seguente codice:

```
Utente utente1 = new Utente();
utente1.Nome = "Mario";
utente1.Cognome = "Rossi";
utente1.Email = "mario.rossi@example.com";
utente1.Password = "password123";
utente1.Registra();
```

In questo esempio, si crea un oggetto "Utente" chiamato "utente1" e si impostano i suoi dati con i valori specificati. Successivamente, si chiama il metodo "Registra()" dell'oggetto "utente1" per registrare l'utente nel sistema.

ACCESSIBILITA' DELLE CLASSI

In C#, le classi possono avere diversi livelli di accessibilità. Questi livelli di accessibilità indicano quali parti del codice possono accedere alla classe e ai suoi membri.

Ecco le diverse tipologie di classe in C#:

1. **Public**: le classi pubbliche sono accessibili da qualsiasi parte del codice, sia all'interno che all'esterno dell'assembly. La maggior parte delle classi dovrebbe essere pubblica, in modo che possa essere utilizzata da altre parti del codice.

   Esempio di classe pubblica:

   ```
   public class MiaClasse
   {
       // Membri della classe
   }
   ```

2. **Private**: le classi private sono accessibili solo all'interno dell'assembly in cui sono definite. Queste classi vengono utilizzate per incapsulare la logica interna e nascondere i dettagli di implementazione alle parti esterne del codice.

   Esempio di classe privata:

   ```
   private class MiaClasse
   {
       // Membri della classe
   }
   ```

3. **Protected**: le classi protette sono accessibili solo dalla classe stessa e dalle sue classi derivate. Questo livello di accessibilità viene utilizzato soprattutto in contesti di ereditarietà, per creare gerarchie di classi e permettere alle classi derivate di accedere ai membri protetti della classe base.

   Esempio di classe protetta:

   ```
   protected class MiaClasse
   {
       // Membri della classe
   }
   ```

4. **Internal**: le classi interne sono accessibili solo all'interno dell'assembly in cui sono definite. Questo livello di accessibilità viene utilizzato per incapsulare la logica interna e nascondere i dettagli di implementazione alle parti esterne del codice, ma consentendo l'accesso alle parti del codice all'interno dello stesso assembly.

   Esempio di classe interna:

   ```
   internal class MiaClasse
   {
       // Membri della classe
   }
   ```

5. **Protected Internal**: le classi protette interne sono accessibili all'interno dell'assembly in cui sono definite e dalle sue classi derivate. Questo livello di accessibilità viene utilizzato soprattutto in contesti di ereditarietà e di incapsulamento di logiche interne.

   Esempio di classe protetta interna:

   ```
   protected internal class MiaClasse
   {
       // Membri della classe
   }
   ```

In generale, si consiglia di utilizzare il livello di accessibilità più restrittivo possibile per incapsulare la logica interna e garantire la sicurezza e la stabilità del codice.

METODI

In C#, un **metodo** è una funzione definita all'interno di una classe che esegue un'operazione specifica e può restituire un valore o essere di tipo "void". Per creare un metodo in C#, si possono seguire i seguenti passaggi:

1. Definire il nome del metodo e i suoi parametri: il nome del metodo dovrebbe descrivere l'operazione che il metodo esegue. I parametri sono opzionali e servono per passare dei valori al metodo.
2. Definire il tipo di valore restituito dal metodo: se il metodo restituisce un valore, è necessario specificare il tipo di valore restituito dal metodo. Se il metodo non restituisce nulla, si può usare la parola chiave "void".
3. Implementare il codice del metodo: questo è il cuore del metodo, dove si scrive il codice che esegue l'operazione desiderata.

Ecco un esempio di come creare un metodo in C#:

```
public class Calcolatrice
{
    // Metodo che somma due numeri
    public int Somma(int a, int b)
    {
        int risultato = a + b;
        return risultato;
    }
}
```

In questo esempio, abbiamo creato un metodo chiamato "Somma" che prende due parametri di tipo "`int`" e restituisce la somma dei due numeri passati come parametri. Il tipo di valore restituito dal metodo è "`int`".

Per utilizzare il metodo, dobbiamo creare un'istanza della classe "Calcolatrice" e chiamare il metodo su quest'istanza:

```
Calcolatrice calcolatrice = new Calcolatrice();
int risultato = calcolatrice.Somma(3, 4); // risultato = 7
```

In questo esempio, abbiamo creato un'istanza della classe "Calcolatrice" con la riga di codice `Calcolatrice calcolatrice = new Calcolatrice();`. Poi abbiamo chiamato il metodo "Somma" su quest'istanza con la riga di codice `int risultato = calcolatrice.Somma(3, 4);`, passando due numeri come parametri. Il metodo "Somma" esegue l'operazione desiderata e restituisce il risultato, che viene salvato nella variabile "risultato".

COSTRUTTORI

In programmazione, un **costruttore** è un metodo speciale di una classe che viene invocato automaticamente al momento della creazione di un'istanza della classe stessa. Il costruttore ha il compito di inizializzare i campi dati dell'oggetto appena creato, in modo che siano pronti per l'utilizzo.

In C#, un costruttore ha lo stesso nome della classe e può avere uno o più parametri. Un costruttore può essere creato in diverse forme a seconda delle esigenze della classe.

Ecco un esempio di costruttore in C#:

```
public class Persona
{
    public string Nome { get; set; }
    public string Cognome { get; set; }
    public int AnnoDiNascita { get; set; }

    // Costruttore della classe Persona
    public Persona(string nome, string cognome, int annoDiNascita)
    {
        Nome = nome;
        Cognome = cognome;
        AnnoDiNascita = annoDiNascita;
    }

    public int CalcolaEta()
    {
        return DateTime.Now.Year - AnnoDiNascita;
    }
}

// Creazione di un'istanza della classe Persona
Persona persona1 = new Persona("Mario", "Rossi", 1980);
```

In questo esempio, abbiamo una classe "Persona" con tre proprietà: "Nome", "Cognome" e "AnnoDiNascita". Abbiamo definito anche un metodo "CalcolaEta()" che restituisce l'età della persona.

Nel costruttore della classe, abbiamo creato tre parametri: "nome", "cognome" e "annoDiNascita". Quando creiamo un'istanza della classe Persona con la riga di codice `Persona persona1 = new Persona("Mario", "Rossi", 1980);`, il costruttore viene invocato automaticamente e inizializza le proprietà della classe con i valori passati come argomenti.

In questo modo, abbiamo un'istanza della classe Persona con i campi dati già inizializzati e possiamo utilizzare i metodi definiti nella classe su quest'istanza.

ESEMPIO CALCOLATORE

Ecco un esempio di codice in C# per un semplice calcolatore che utilizza classi e metodi per eseguire le operazioni matematiche:

```
using System;

namespace CalculatorApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Welcome to the calculator app!");
            Console.WriteLine("Please enter the first number:");
            double num1 = double.Parse(Console.ReadLine());

            Console.WriteLine("Please enter the second number:");
            double num2 = double.Parse(Console.ReadLine());

            Calculator calculator = new Calculator();
            Console.WriteLine("Addition result: " + calculator.Add(num1, num2));
            Console.WriteLine("Subtraction result: " + calculator.Subtract(num1, num
2));
            Console.WriteLine("Multiplication result: " + calculator.Multiply(num1, nu
m2));
            Console.WriteLine("Division result: " + calculator.Divide(num1, num2));
        }
    }

    class Calculator
    {
        public double Add(double num1, double num2)
        {
            return num1 + num2;
        }

        public double Subtract(double num1, double num2)
        {
            return num1 - num2;
        }

        public double Multiply(double num1, double num2)
        {
            return num1 * num2;
        }

        public double Divide(double num1, double num2)
        {
            if (num2 == 0)
            {
                throw new DivideByZeroException("Cannot divide by zero");
            }
            return num1 / num2;
        }
```

```
    }
}
```

In questo esempio, il programma chiede all'utente di inserire due numeri, quindi crea un'istanza della classe `Calculator` e utilizza i suoi metodi per eseguire le operazioni matematiche richieste. La classe `Calculator` contiene i metodi per l'addizione, la sottrazione, la moltiplicazione e la divisione dei numeri. Se viene inserito uno zero come secondo numero durante la divisione, il metodo `Divide` lancia un'eccezione di tipo `DivideByZeroException`.

ALGORITMI COMUNEMENTE USATI NELLA PROGRAMMAZIONE:

In generale, un **algoritmo** in C# è una serie di istruzioni o passaggi logici e matematici che vengono eseguiti per risolvere un determinato problema o eseguire una determinata operazione. Essi possono essere utilizzati in diversi contesti, come ad esempio nell'elaborazione di dati, nella matematica computazionale, nella grafica, nell'intelligenza artificiale e in molti altri campi.

1. **Algoritmo di ordinamento**: Gli algoritmi di ordinamento sono utilizzati per organizzare un insieme di dati in un certo ordine. Ci sono molti algoritmi di ordinamento comuni come il ***Bubble Sort***, Insertion Sort, Merge Sort e Quick Sort.
2. **Algoritmo di ricerca**: Gli algoritmi di ricerca sono utilizzati per trovare un determinato valore all'interno di un insieme di dati. Ci sono molti algoritmi di ricerca comuni come la ricerca lineare e la ricerca binaria.
3. **Algoritmo di compressione**: Gli algoritmi di compressione sono utilizzati per ridurre la quantità di spazio necessario per archiviare un insieme di dati. Alcuni esempi di algoritmi di compressione includono GZip, LZ77 e Huffman.
4. **Algoritmo di apprendimento automatico**: L'apprendimento automatico è un'area della programmazione che utilizza algoritmi per far imparare a un computer a risolvere determinati problemi o compiti. Alcuni esempi di algoritmi di apprendimento automatico includono reti neurali, alberi decisionali e regressione logistica.
5. **Algoritmo di crittografia**: Gli algoritmi di crittografia sono utilizzati per proteggere i dati sensibili da accessi non autorizzati. Alcuni esempi di algoritmi di crittografia includono ***AES***, ***RSA*** e Blowfish.

6. **Algoritmo di ricerca del percorso**: Gli algoritmi di ricerca del percorso sono utilizzati per trovare il percorso più breve tra due punti in un grafo. Alcuni esempi di algoritmi di ricerca del percorso includono Dijkstra e l'algoritmo di Bellman-Ford.
7. **Algoritmo di clustering**: Gli algoritmi di clustering sono utilizzati per raggruppare insieme oggetti simili. Alcuni esempi di algoritmi di clustering includono il k-means clustering e il clustering gerarchico.

Algoritmo di ordinamento a bolle (Bubble Sort)

```
static void BubbleSort(int[] arr)
{
    int n = arr.Length;
    bool swapped;
    do
    {
        swapped = false;
        for (int i = 1; i < n; i++)
        {
            if (arr[i - 1] > arr[i])
            {
                int temp = arr[i];
                arr[i] = arr[i - 1];
                arr[i - 1] = temp;
                swapped = true;
            }
        }
        n--;
    } while (swapped);
}
```

Algoritmo di ordinamento per inserimento (Insertion Sort)

```
static void InsertionSort(int[] arr)
{
    for (int i = 1; i < arr.Length; i++)
    {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
```

```
    }
}
```

Algoritmo di ordinamento per selezione (Selection Sort)

```
static void SelectionSort(int[] arr)
{
    for (int i = 0; i < arr.Length - 1; i++)
    {
        int minIndex = i;
        for (int j = i + 1; j < arr.Length; j++)
        {
            if (arr[j] < arr[minIndex])
            {
                minIndex = j;
            }
        }
        if (minIndex != i)
        {
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}
```

Algoritmo di ricerca binaria (Binary Search)

```
static int BinarySearch(int[] arr, int value)
{
    int left = 0;
    int right = arr.Length - 1;
    while (left <= right)
    {
        int mid = left + (right - left) / 2;
        if (arr[mid] == value)
        {
            return mid;
        }
        else if (arr[mid] < value)
        {
            left = mid + 1;
        }
        else
        {
            right = mid - 1;
        }
    }
    return -1;
}
```

Algoritmo di visita in ampiezza (Breadth-First Search)

```
public class Node
{
    public int Value;
    public List<Node> Neighbors = new List<Node>();
}

public static void BreadthFirstSearch(Node start)
{
    Queue<Node> queue = new Queue<Node>();
    HashSet<Node> visited = new HashSet<Node>();
    queue.Enqueue(start);
    visited.Add(start);
    while (queue.Count > 0)
    {
        Node node = queue.Dequeue();
        Console.WriteLine(node.Value);
        foreach (Node neighbor in node.Neighbors)
        {
            if (!visited.Contains(neighbor))
            {
                queue.Enqueue(neighbor);
                visited.Add(neighbor);
            }
        }
    }
}
```

Algoritmo di Monte Carlo

```
using System;

class MonteCarloSimulation
{
    static void Main()
    {
        int n = 1000000; // Numero di simulazioni da eseguire
        int m = 0; // Contatore per il numero di simulazioni che rientrano nel cerchio
        Random rand = new Random();

        for (int i = 0; i < n; i++)
        {
            double x = rand.NextDouble(); // Genera un numero casuale tra 0 e 1 per la
coordinata x
            double y = rand.NextDouble(); // Genera un numero casuale tra 0 e 1 per la
coordinata y

            if ((x * x + y * y) <= 1) // Controlla se il punto è all'interno del cerch
io
            {
                m++; // Incrementa il contatore se il punto è all'interno del cerchio
            }
```

```
        }

        double pi = 4.0 * m / n; // Calcola il valore di pi

        Console.WriteLine("Valore stimato di pi: " + pi);
        Console.ReadKey();
    }
}
```