|  |  |
| --- | --- |
| **Var** | Từ khóa **var**: Trong C#, từ khóa var được sử dụng để khai báo biến mà không cần chỉ định kiểu dữ liệu cụ thể. Trình biên dịch sẽ tự động suy luận kiểu dữ liệu của biến dựa trên giá trị khởi tạo của nó.    var không phải là một kiểu dữ liệu: var chỉ là một từ khóa để trình biên dịch suy luận kiểu dữ liệu, không phải là kiểu dữ liệu thực sự. |
| ****Dynamic**** | Dynamic type trong C# là một kiểu dữ liệu đặc biệt cho phép các biến được khai báo mà không cần xác định kiểu dữ liệu cụ thể tại thời điểm biên dịch. Kiểu dữ liệu của biến sẽ được xác định tại thời điểm chạy chương trình (runtime). Dưới đây là giải thích chi tiết và ví dụ minh họa.  **Không kiểm tra kiểu tại thời điểm biên dịch:**  Trình biên dịch không kiểm tra kiểu dữ liệu của biến dynamic khi biên dịch. Việc kiểm tra và xác định kiểu dữ liệu diễn ra tại runtime.  **Hành vi giống kiểu object:**  Biến dynamic có hành vi tương tự như kiểu object, nhưng khác biệt ở chỗ các phép toán và truy cập thành viên được kiểm tra tại runtime thay vì compile time.  **Thay đổi kiểu dữ liệu tại runtime:**  Kiểu dữ liệu của biến dynamic có thể thay đổi dựa trên giá trị được gán cho biến tại runtime.  **Dùng GetType() để lấy kiểu thực sự:**  Bạn có thể sử dụng phương thức GetType() để lấy kiểu dữ liệu thực sự của biến dynamic tại runtime.  **Truyền tham số kiểu dynamic trong phương thức:**  Các phương thức có thể chấp nhận tham số kiểu dynamic, cho phép chúng nhận bất kỳ kiểu dữ liệu nào tại runtime. |
| **ref** | Trong **C#**, **ref** là từ khóa dùng để truyền tham số theo tham chiếu, cho phép phương thức thay đổi giá trị của tham số và những thay đổi này sẽ được duy trì bên ngoài phương thức.  **ref** biến tham số thành một bí danh (**alias**) cho đối số, cho phép phương thức thay đổi giá trị của biến gốc.  Đối số truyền vào tham số ref phải được khởi tạo trước khi được truyền vào phương thức. |
| **out** | Trong C#, **out** là từ khóa dùng để truyền tham số theo tham chiếu, cho phép phương thức thay đổi giá trị của tham số và những thay đổi này sẽ được duy trì bên ngoài phương thức.   * **out** cũng biến tham số thành một bí danh cho đối số, nhưng khác với **ref**, đối số truyền vào tham số **out** không cần được khởi tạo trước khi truyền vào. * Tham số **out** phải được gán giá trị bên trong phương thức trước khi phương thức kết thúc. |
| **params** | Trong C#, từ khóa **params** cho phép bạn truyền một số lượng biến đổi các tham số có cùng kiểu vào một phương thức dưới dạng một tham số logic duy nhất. Điều này rất hữu ích khi bạn không biết trước số lượng tham số sẽ được truyền vào phương thức. Dưới đây là một giải thích chi tiết về cách sử dụng từ khóa **params**.   1. ****Truyền nhiều tham số****: Bạn có thể truyền một số lượng biến đổi các tham số có cùng kiểu vào một phương thức. 2. ****Dạng mảng****: Các tham số này sẽ được xử lý như một mảng một chiều trong phương thức. 3. ****Cách truyền tham số****: Tham số có thể được truyền dưới dạng mảng mạnh kiểu hoặc một danh sách các giá trị cách nhau bằng dấu phẩy. 4. ****Hạn chế****:    * Kiểu tham số phải là một mảng một chiều.    * Không được có thêm bất kỳ tham số nào sau tham số **params** trong khai báo phương thức.    * Chỉ được phép sử dụng một từ khóa **params** trong một khai báo phương thức.      * ****Từ khóa** params**: Cho phép truyền một số lượng biến đổi các tham số có cùng kiểu vào một phương thức. * ****Dạng mảng một chiều****: Các tham số được truyền vào sẽ được xử lý như một mảng một chiều. * ****Hạn chế****: Phải là tham số cuối cùng trong danh sách tham số, chỉ được phép có một tham số **params** trong mỗi phương thức. |
| Namespace | Namespace (không gian tên) trong C# là một cách để tổ chức và nhóm các lớp, giao diện (interfaces), cấu trúc (structs), và các kiểu dữ liệu khác lại với nhau nhằm tránh xung đột tên và quản lý mã nguồn một cách hiệu quả. Chúng giúp kiểm soát phạm vi của các phương thức và lớp trong các dự án lập trình lớn hơn.   1. ****Tránh xung đột tên (Name Conflict)****: Khi làm việc trên các dự án lớn hoặc sử dụng các thư viện bên ngoài, namespace giúp tránh các xung đột tên giữa các lớp hoặc phương thức có cùng tên. 2. ****Tổ chức mã nguồn (Code Organization)****: Giúp tổ chức mã nguồn một cách có cấu trúc và dễ quản lý hơn. 3. ****Kiểm soát phạm vi (Scope Control)****: Giúp kiểm soát phạm vi của các phương thức và lớp trong các dự án lập trình lớn hơn. |
| Tuples | là một cấu trúc dữ liệu có thể chứa một nhóm các phần tử với các kiểu dữ liệu khác nhau. Chúng được sử dụng để lưu trữ nhiều giá trị liên quan mà không cần phải định nghĩa một lớp hoặc cấu trúc mới. Tuples có thể chứa từ một đến nhiều phần tử và các phần tử này có thể có các kiểu dữ liệu khác nhau. |
| Discards | một tính năng cho phép bạn bỏ qua các giá trị mà bạn không cần sử dụng. Điều này giúp mã nguồn của bạn ngắn gọn và dễ đọc hơn bằng cách không phải đặt tên cho các biến mà bạn sẽ không sử dụng.  Discards được biểu diễn bằng ký tự gạch dưới (\_). Bạn có thể sử dụng discards trong nhiều ngữ cảnh khác nhau, bao gồm việc hủy bỏ các giá trị trả về từ một phương thức, các phần tử trong tuples, các tham số của các sự kiện, và nhiều trường hợp khác. Discards trong Tuples Khi một phương thức trả về một tuple, bạn có thể sử dụng discards để bỏ qua các phần tử mà bạn không cần.   Discards trong Out Parameters Khi sử dụng từ khóa out để nhận nhiều giá trị trả về từ một phương thức, bạn có thể sử dụng discards để bỏ qua các giá trị không cần thiết. |
| Pattern Matching | (so khớp mẫu) là một tính năng mạnh mẽ trong C# được giới thiệu từ phiên bản C# 7.0 và đã được mở rộng trong các phiên bản sau đó. Pattern Matching cho phép bạn kiểm tra và trích xuất các giá trị từ các đối tượng dựa trên cấu trúc của chúng, tương tự như cách bạn sử dụng các câu lệnh điều kiện. Nó làm cho mã nguồn của bạn ngắn gọn và dễ đọc hơn, đặc biệt là khi làm việc với các kiểu dữ liệu phức tạp. **Các loại Pattern Matching trong C#** **Type Pattern (Kiểu mẫu):** Sử dụng để kiểm tra kiểu của một đối tượng và nếu đúng, thực hiện việc chuyển đổi kiểu một cách an toàn.    **Constant Pattern (Hằng mẫu):** Sử dụng để so khớp giá trị của một biến với một hằng số.    **Relational Pattern (Mẫu quan hệ):** Được giới thiệu trong C# 9.0, cho phép so khớp các biểu thức quan hệ như lớn hơn, nhỏ hơn.    **Logical Pattern (Mẫu logic):** Cũng được giới thiệu trong C# 9.0, cho phép kết hợp các mẫu với các toán tử logic như and, or, not.    **Property Pattern (Mẫu thuộc tính):** Cho phép so khớp các thuộc tính của một đối tượng.    **Positional Pattern (Mẫu vị trí):** Được sử dụng với các kiểu dữ liệu như tuples và records để so khớp các thành phần của chúng theo vị trí.   Sử dụng Pattern Matching trong switch expressions Pattern Matching cũng được tích hợp vào các biểu thức switch, làm cho chúng mạnh mẽ hơn. |
| when | Nó cho phép bạn chỉ định thêm các điều kiện    · case int n: Đây là mẫu kiểu (type pattern) kiểm tra xem number có phải là số nguyên (int) hay không. Nếu đúng, giá trị của number sẽ được gán cho biến n.  · when (n > 0): Điều kiện bổ sung kiểm tra xem n có lớn hơn 0 hay không.  · Nếu cả hai điều kiện đều thỏa mãn, câu lệnh Console.WriteLine($"{n} is positive."); sẽ được thực thi và in ra màn hình 42 is positive.. |
| Null-Condition Operators | (toán tử điều kiện null) trong C# là một cú pháp tiện lợi giúp bạn làm việc với các đối tượng có thể có giá trị null mà không cần phải kiểm tra thủ công từng trường hợp. Nó giúp giảm thiểu lỗi NullReferenceException bằng cách chỉ thực hiện các hành động tiếp theo nếu đối tượng không phải là null.  **Toán tử ?. (Null-conditional member access):** Sử dụng để truy cập các thành viên của một đối tượng (thuộc tính, phương thức, hoặc chỉ mục) một cách an toàn mà không gặp phải lỗi nếu đối tượng đó là null. |