|  |  |
| --- | --- |
| 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.  Truyền nhiều tham số: Bạn có thể truyền một số lượng biến đổi các tham số có cùng kiểu vào một phương thức.  Dạng mảng: Các tham số này sẽ được xử lý như một mảng một chiều trong phương thức.  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.  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.  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.  Tổ chức mã nguồn (Code Organization): Giúp tổ chức mã nguồn một cách có cấu trúc và dễ quản lý hơn.  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    Toán tử ?[] (Null-conditional index access): Sử dụng để truy cập các phần tử của mảng hoặc các bộ sưu tập theo chỉ số một cách an toàn. |
| is | Toán tử is được sử dụng để kiểm tra xem một đối tượng có phải là một thể hiện của một kiểu cụ thể hay không. Nó trả về true nếu đối tượng là một thể hiện của kiểu cụ thể đó, ngược lại trả về false. |
| as | Toán tử as được sử dụng để thực hiện ép kiểu an toàn. Nó cố gắng ép kiểu một đối tượng sang kiểu cụ thể. Nếu ép kiểu thành công, nó trả về đối tượng đã được ép kiểu; nếu không, nó trả về null mà không ném ra ngoại lệ. |
| object | Trong C#, object là kiểu dữ liệu cơ bản nhất và là kiểu cơ sở (base type) của tất cả các kiểu dữ liệu khác. Mọi kiểu dữ liệu trong C#, bao gồm các kiểu giá trị (value types) và các kiểu tham chiếu (reference types), đều được dẫn xuất từ object. Điều này có nghĩa là bất kỳ kiểu dữ liệu nào cũng có thể được gán cho một biến có kiểu object. Đặc điểm của kiểu object  * **Kiểu cơ sở của tất cả các kiểu:** Mọi kiểu dữ liệu trong C# đều dẫn xuất từ object. Điều này cho phép bạn lưu trữ bất kỳ giá trị nào trong một biến có kiểu object. * **Hỗ trợ boxing và unboxing:** Khi một kiểu giá trị được gán cho một biến kiểu object, quá trình "boxing" xảy ra, chuyển đổi giá trị đó thành một đối tượng trên heap. Khi một giá trị kiểu object được chuyển đổi ngược lại thành một kiểu giá trị, quá trình "unboxing" xảy ra. * **Khả năng đa hình (polymorphism):** Sử dụng kiểu object cho phép bạn viết mã có tính tổng quát và linh hoạt hơn, có thể xử lý các đối tượng của nhiều kiểu khác nhau. |
| Boxing | **Boxing** là quá trình chuyển đổi một kiểu giá trị (value type) thành một đối tượng (object) của kiểu System.Object hoặc bất kỳ giao diện (interface) nào mà kiểu giá trị đó triển khai. Boxing là một quá trình ngầm định và xảy ra tự động khi một giá trị kiểu giá trị cần được chuyển đổi thành kiểu đối tượng. |
| Delegate | **Delegate** trong C# là một kiểu dữ liệu đặc biệt cho phép bạn tham chiếu đến các phương thức. Delegate giống như một con trỏ hàm trong C/C++ nhưng an toàn kiểu hơn. Delegate cho phép bạn gọi các phương thức thông qua đối tượng delegate, và nó rất hữu ích trong việc thực hiện các phương thức gọi lại (callbacks), sự kiện (events), và các phương thức xử lý không đồng bộ (asynchronous methods).  - nói cách khác delegate có thể giúp truy cập tới một phương thức nào đó thông qua bằng cách tạo một đối tượng có phương thức trả về là delegate rồi từ đối tượng đó có thể sử dụng được các phương thức khác.      Delegate tích hợp sẵn trong .NET  C# cung cấp một số delegate tích hợp sẵn trong thư viện System, bao gồm:  **Action**: Delegate không trả về giá trị, có thể nhận 0 hoặc nhiều tham số.    **Func:** Delegate có thể trả về một giá trị, với tối đa 16 tham số đầu vào.    **Predicate:** Delegate đại diện cho một phương thức trả về kiểu bool và nhận một tham số.    Delegate thường được sử dụng để xử lý sự kiện trong C#. Sự kiện là một cơ chế để thông báo cho các đối tượng khác khi một hành động cụ thể xảy ra. |
| **Func** | là một delegate tổng quát được sử dụng để đại diện cho một phương thức có thể trả về một giá trị. Nó có thể nhận từ 0 đến 16 tham số đầu vào. Tham số cuối cùng của Func luôn là kiểu trả về. |
| **Action** | **Action** là một delegate tổng quát được sử dụng để đại diện cho một phương thức không trả về giá trị (void). Nó có thể nhận từ 0 đến 16 tham số đầu vào. |
| => | Dấu => trong C# được gọi là **lambda operator** (toán tử lambda) và nó được sử dụng để định nghĩa các biểu thức lambda (lambda expressions). Biểu thức lambda là một cách ngắn gọn để viết các phương thức ẩn danh, và chúng thường được sử dụng trong LINQ (Language Integrated Query) cũng như các delegate và sự kiện |
| **Events** | **Events** (sự kiện) trong C# là một cơ chế cho phép các đối tượng thông báo cho các đối tượng khác khi một hành động cụ thể xảy ra. Events thường được sử dụng để thực hiện mô hình phát-nghe (publish-subscribe), trong đó một đối tượng phát (publisher) phát ra sự kiện và các đối tượng nghe (subscribers) đăng ký để xử lý sự kiện đó. |