## **Generics**

Generics provide a mechanism for writing code that operates on data of different types, but in a type-safe manner. Unlike inheritance, which expresses reusability through a common base type, generics use a "template" approach with "placeholder" types.Generik menyediakan mekanisme untuk menulis kode yang beroperasi pada data dengan tipe yang berbeda, namun tetap aman secara tipe. Berbeda dengan pewarisan, yang mengekspresikan kemudahan penggunaan kembali melalui tipe dasar yang sama, generik menggunakan pendekatan “template” dengan “tipe tempatholder”.

The primary benefits of generics, especially when compared to using object as a general type:Manfaat utama dari generik, terutama jika dibandingkan dengan penggunaan objek sebagai tipe umum:

* **Increased Type Safety:** Errors are caught at compile time, not runtime.Peningkatan Keamanan Tipe: Kesalahan terdeteksi pada saat kompilasi, bukan pada saat runtime.
* **Reduced Casting:** You avoid the need for explicit downcasting, making code cleaner and less error-prone.Pengurangan Konversi Tipe: Anda menghindari kebutuhan untuk melakukan konversi tipe secara eksplisit, sehingga kode menjadi lebih bersih dan kurang rentan terhadap kesalahan.
* **Reduced Boxing/Unboxing:** For value types, generics eliminate the performance overhead associated with boxing and unboxing.Pengurangan Boxing/Unboxing: Untuk tipe nilai, generik menghilangkan beban kinerja yang terkait dengan boxing dan unboxing.

### **Generic Types**

A **generic type** declares **type parameters**—placeholder types that are filled in by the consumer of the generic type, who supplies **type arguments**.Tipe generik mendeklarasikan parameter tipe—tipe tempat penampung yang diisi oleh pengguna tipe generik, yang menyediakan argumen tipe.

Consider our Stack example, now made generic to stack instances of any type T:Pertimbangkan contoh Stack kami, yang kini dibuat generik untuk instance Stack dari tipe apa pun T:

|  |
| --- |
| public class Stack<T> // Declares a type parameter T {  int position;  T[] data = new T[100]; // Array of type T   public void Push(T obj) => data[position++] = obj; // Accepts type T  public T Pop() => data[--position]; // Returns type T } |

When we use Stack<T>, we specify the actual type argument. The CLR then synthesizes a specific type on the fly at runtime:Ketika kita menggunakan Stack<T>, kita menentukan argumen tipe yang sebenarnya. CLR kemudian mensintesis tipe spesifik secara dinamis pada saat runtime:

|  |
| --- |
| var stack = new Stack<int>(); // Type argument 'int' fills in for 'T' stack.Push(5); stack.Push(10); int x = stack.Pop(); // x is 10 int y = stack.Pop(); // y is 5  // stack.Push("hello"); // Compile-time error: Cannot push a string onto a Stack<int> |

In this example, Stack<int> is a **closed type** (all type parameters have been filled in), while Stack<T> (the class definition itself) is an **open type**. At runtime, all generic type instances are closed.Dalam contoh ini, Stack<int> adalah tipe tertutup (semua parameter tipe telah diisi), sedangkan Stack<T> (definisi kelas itu sendiri) adalah tipe terbuka. Pada saat runtime, semua instansi tipe generik menjadi tertutup.

You cannot instantiate an open generic type directly:Anda tidak dapat menginstansiasi tipe generik terbuka secara langsung:

|  |
| --- |
| // var stack = new Stack<>; // Illegal: What is T? |

However, it's legal within a class or method that itself defines T as a type parameter (e.g., in a generic method or a nested generic class).Namun, hal ini diperbolehkan di dalam kelas atau metode yang sendiri mendefinisikan T sebagai parameter tipe (misalnya, dalam metode generik atau kelas generik bersarang).

### **Why Generics Exist**

Generics address the challenges of writing reusable code that is both flexible and type-safe.Generik mengatasi tantangan dalam menulis kode yang dapat digunakan kembali yang fleksibel dan aman tipe.

* **Code Duplication (Hardcoded types):** Without generics, if you needed a stack for int, string, DateTime, etc., you'd have to write separate IntStack, StringStack, DateTimeStack classes, leading to massive code duplication.Duplikasi Kode (Tipe yang Dihardcode): Tanpa generik, jika Anda membutuhkan tumpukan untuk int, string, DateTime, dll., Anda harus menulis kelas IntStack, StringStack, DateTimeStack secara terpisah, yang mengakibatkan duplikasi kode yang besar.
* **Lack of Type Safety and Performance Issues (object type):** While using object (as in ObjectStack from earlier) allows for generality, it introduces problems:Kekurangan Keamanan Tipe dan Masalah Kinerja (tipe objek): Meskipun penggunaan objek (seperti ObjectStack sebelumnya) memungkinkan fleksibilitas, hal ini menimbulkan masalah:
  + **No Compile-time Type Checking:** You could accidentally push a string onto a stack intended for integers, leading to a runtime error when trying to downcast.Tidak Ada Pemeriksaan Tipe pada Waktu Kompilasi: Anda dapat secara tidak sengaja memasukkan string ke dalam tumpukan yang dimaksudkan untuk bilangan bulat, yang dapat menyebabkan kesalahan waktu berjalan saat mencoba melakukan downcast.
  + **Boxing/Unboxing Overhead:** Storing value types (like int) as object requires boxing, which allocates memory on the heap and incurs a performance penalty. Unboxing is also required to retrieve the original value type.Biaya Boxing/Unboxing: Menyimpan tipe data nilai (seperti int) sebagai objek memerlukan proses boxing, yang mengalokasikan memori di heap dan menimbulkan penalti kinerja. Proses unboxing juga diperlukan untuk mengambil kembali tipe data nilai asli.

Generics provide the best of both worlds: a single, general implementation (Stack<T>) that works for all element types, combined with compile-time type safety and performance benefits (no boxing/unboxing for value types) when specialized to a specific type (Stack<int>).Generik menawarkan yang terbaik dari kedua dunia: implementasi tunggal dan umum (Stack<T>) yang berfungsi untuk semua jenis elemen, dikombinasikan dengan keamanan tipe pada waktu kompilasi dan manfaat kinerja (tidak ada boxing/unboxing untuk jenis nilai) saat dispesialisasikan ke jenis tertentu (Stack<int>).

## **Generic Methods**

A **generic method** declares type parameters within its own signature. This allows you to write methods that can operate on different types.Metode generik mendeklarasikan parameter tipe di dalam tanda tangannya sendiri. Hal ini memungkinkan Anda untuk menulis metode yang dapat beroperasi pada berbagai jenis data.

|  |
| --- |
| static void Swap<T>(ref T a, ref T b) // Declares type parameter T {  T temp = a;  a = b;  b = temp; } |

Typically, the compiler can **implicitly infer the type arguments** when you call a generic method, so you don't need to specify them explicitly:Secara umum, compiler dapat secara implisit menentukan argumen tipe saat Anda memanggil metode generik, sehingga Anda tidak perlu menentukannya secara eksplisit:

|  |
| --- |
| int x = 5; int y = 10; Swap(ref x, ref y); // Compiler infers T is int |

If there's ambiguity, you can explicitly provide the type arguments: Swap<int>(ref x, ref y);.Jika ada ambiguitas, Anda dapat secara eksplisit menyediakan argumen tipe: Swap<int>(ref x, ref y);.

**Important Note:** Only **methods** and **types** (classes, structs, interfaces, delegates) can introduce new type parameters using the angle bracket syntax (<T>). Other members like properties, indexers, events, fields, and constructors cannot introduce their own type parameters, though they can use type parameters declared by their enclosing generic type.Catatan Penting: Hanya metode dan tipe (kelas, struktur, antarmuka, delegasi) yang dapat memperkenalkan parameter tipe baru menggunakan sintaks kurung sudut (<T>). Anggota lain seperti properti, indeks, peristiwa, bidang, dan konstruktor tidak dapat memperkenalkan parameter tipe mereka sendiri, meskipun mereka dapat menggunakan parameter tipe yang dideklarasikan oleh tipe generik yang mengelilinginya.

|  |
| --- |
| public struct Nullable<T> // T is from the struct declaration {  public T Value { get; } // Uses T } |

## 

## **Declaring Type Parameters**

* Type parameters can be introduced in class, struct, interface, delegate, and method declarations.Parameter tipe dapat diperkenalkan dalam deklarasi kelas, struktur, antarmuka, delegasi, dan metode.
* A generic type or method can have **multiple type parameters**:Sebuah tipe atau metode generik dapat memiliki beberapa parameter tipe:

|  |
| --- |
| class Dictionary<TKey, TValue> { ... } // Instantiation: var myDict = new Dictionary<string, int>(); |

* Generic types and methods can be **overloaded** as long as the number of type parameters (their "arity") differs. A, A<T>, and A<T1, T2> are distinct.Tipe dan metode generik dapat di-overload selama jumlah parameter tipe (arbitrasinya) berbeda. A, A<T>, dan A<T1, T2> adalah berbeda.

**Convention:**

* For single type parameters, use T.Untuk parameter tipe tunggal, gunakan T.
* For multiple type parameters, prefix with T and use descriptive names (e.g., TKey, TValue).Untuk parameter tipe ganda, tambahkan awalan T dan gunakan nama yang deskriptif (misalnya, TKey, TValue).

## **typeof and Unbound Generic Types**

While open generic types (like List<T>) are compiled and then closed at runtime, you can represent an **unbound generic type** purely as a System.Type object using the typeof operator. This is primarily used with Reflection.Meskipun tipe generik terbuka (seperti List<T>) dikompilasi dan kemudian ditutup pada waktu runtime, Anda dapat mewakili tipe generik yang tidak terikat secara murni sebagai objek System.Type menggunakan operator typeof. Hal ini terutama digunakan bersama dengan Reflection.

|  |
| --- |
| Type listType = typeof(List<>); // Unbound generic type Type dictType = typeof(Dictionary<,>); // Use commas to indicate multiple type parameters |

You can also use typeof with closed types (typeof(List<int>)) or with generic type parameters themselves within a generic context (typeof(T) inside List<T>).Anda juga dapat menggunakan typeof dengan tipe tertutup (typeof(List<int>)) atau dengan parameter tipe generik itu sendiri dalam konteks generik (typeof(T) di dalam List<T>).

## **The default Generic Value**

The default keyword is used to get the default value for a generic type parameter T.Kata kunci default digunakan untuk mendapatkan nilai default untuk parameter tipe generik T.

* If T is a **reference type**, default(T) is null.Jika T adalah tipe referensi, default(T) adalah null.
* If T is a **value type**, default(T) is the result of bitwise-zeroing its fields (e.g., 0 for int, false for bool).Jika T adalah tipe nilai, default(T) adalah hasil dari mengosongkan bit-bit bidangnya (misalnya, 0 untuk int, false untuk bool).

|  |
| --- |
| static void Zap<T>(T[] array) {  for (int i = 0; i < array.Length; i++)  array[i] = default(T); // Sets each element to its default value } |

From C# 7.1, you can often omit (T) if the compiler can infer the type: array[i] = default;.Mulai dari C# 7.1, Anda sering dapat menghilangkan (T) jika compiler dapat menebak tipe: array[i] = default;.

## **Generic Constraints**

By default, any type can be substituted for a type parameter. **Constraints** are used to restrict the types that can be supplied as type arguments. This is crucial because it allows you to call methods or access properties on the generic type parameter that would otherwise be unknown.Secara default, jenis apa pun dapat digunakan sebagai pengganti parameter jenis. Batasan digunakan untuk membatasi jenis yang dapat disediakan sebagai argumen jenis. Hal ini sangat penting karena memungkinkan Anda memanggil metode atau mengakses properti pada parameter jenis generik yang sebaliknya tidak diketahui.

Possible constraints:

* where T : base-class: T must be or derive from base-class.
* where T : interface: T must implement interface.
* where T : class: T must be a reference type.
* where T : struct: T must be a non-nullable value type.
* where T : unmanaged (C# 7.3+): T must be an unmanaged type (simple value type or struct recursively free of reference types).
* where T : new(): T must have a public parameterless constructor.
* where U : T: U must be or derive from T (a "naked type" constraint).
* where T : notnull (C# 8+): T must be a non-nullable value type or a non-nullable reference type.

Constraints enable operations that would otherwise be impossible. For instance, the IComparable interface is commonly used to constrain generic Max or Sort methods:Batasan memungkinkan operasi yang sebaliknya tidak mungkin dilakukan. Misalnya, antarmuka IComparable sering digunakan untuk membatasi metode Max atau Sort yang generik:

|  |
| --- |
| static T Max<T>(T a, T b) where T : IComparable<T> // Constraint: T must implement IComparable<T> {  return a.CompareTo(b) > 0 ? a : b; // Now can call CompareTo() on 'a' and 'b' } |

This Max method can now be used with any type that implements IComparable<T>, such as int or string.Metode Max ini sekarang dapat digunakan dengan tipe apa pun yang mengimplementasikan antarmuka IComparable<T>, seperti int atau string.

From C# 11, interface constraints (where T : IMyInterface) also enable calling **static virtual/abstract members** on that interface, leading to the concept of "Static Polymorphism" or "Generic Math."Mulai dari C# 11, batasan antarmuka (where T : IMyInterface) juga memungkinkan pemanggilan anggota statis virtual/abstrak pada antarmuka tersebut, yang mengarah pada konsep “Polimorfisme Statis” atau “Matematika Generik.”

## **Subclassing Generic Types**

A generic class can be subclassed. The subclass can:Sebuah kelas generik dapat diwarisi. Subkelas tersebut dapat:

* **Leave the base class's type parameters open:**Biarkan parameter tipe kelas dasar tetap terbuka:

|  |
| --- |
| class Stack<T> { ... } class SpecialStack<T> : Stack<T> { ... } |

* **Close the generic type parameters with a concrete type:Tutup parameter tipe generik dengan tipe konkret:**

|  |
| --- |
| class IntStack : Stack<int> { ... } |

## **Self-Referencing Generic Declarations**Deklarasi Generik yang Merujuk pada Diri Sendiri

A type can refer to itself as the concrete type argument when closing a type parameter, typically used in interfaces like IEquatable<T>:Sebuah tipe dapat merujuk pada dirinya sendiri sebagai argumen tipe konkret saat menutup parameter tipe, biasanya digunakan dalam antarmuka seperti IEquatable<T>:

|  |
| --- |
| public interface IEquatable<T> { bool Equals(T obj); } public class Balloon : IEquatable<Balloon> // Balloon implements IEquatable<Balloon> {  public string Color { get; set; }  public int CC { get; set; }  public bool Equals(Balloon b) // Now 'b' is directly of type Balloon  {  if (b == null) return false;  return b.Color == Color && b.CC == CC;  } } |

## **Static Data in Generic Types**

Static data within a generic type is **unique for each closed type**. This means Count in Bob<int> is separate from Count in Bob<string>.Data statis dalam tipe generik unik untuk setiap tipe tertutup. Artinya, Count dalam Bob<int> terpisah dari Count dalam Bob<string>.

|  |
| --- |
| class Bob<T> { public static int Count; }  Console.WriteLine(++Bob<int>.Count); // 1 Console.WriteLine(++Bob<int>.Count); // 2 Console.WriteLine(++Bob<string>.Count); // 1 Console.WriteLine(++Bob<double>.Count); // 1 |

## **Type Parameters and Conversions**

When performing conversions with generic type parameters (T), the compiler needs to determine the conversion type (numeric, reference, boxing/unboxing, or custom). Since T's exact type is unknown at compile time, ambiguities can arise.Saat melakukan konversi dengan parameter tipe generik (T), compiler perlu menentukan jenis konversi (numerik, referensi, boxing/unboxing, atau kustom). Karena tipe tepat dari T tidak diketahui pada saat kompilasi, ambiguitas dapat timbul.

**Problem with direct casting:**

|  |
| --- |
| StringBuilder Foo<T>(T arg) {  // return (StringBuilder)arg; // Compile-time error: Ambiguous conversion } |

The compiler doesn't know if T is a reference type or a value type that might have a custom conversion to StringBuilder.Penerjemah tidak mengetahui apakah T adalah tipe referensi atau tipe nilai yang mungkin memiliki konversi khusus ke StringBuilder.

**Solutions:**

1. **Use the as operator:** This operator only performs reference or nullable conversions and never custom conversions, making it unambiguous. It returns null on failure.Gunakan operator as: Operator ini hanya melakukan konversi referensi atau konversi nullable dan tidak pernah melakukan konversi kustom, sehingga tidak ambigu. Operator ini mengembalikan null jika terjadi kegagalan.

|  |
| --- |
| StringBuilder Foo<T>(T arg) {  StringBuilder sb = arg as StringBuilder; // OK  if (sb != null) return sb;  return null; } |

1. **Cast to object first:** Conversions to/from object are assumed not to be custom conversions. This resolves the ambiguity.Konversi ke/dari objek dianggap bukan konversi kustom. Hal ini menghilangkan ambiguitas.

|  |
| --- |
| StringBuilder Foo<T>(T arg) {  return (StringBuilder)(object)arg; // OK: First to object, then reference conversion } |

Similarly for unboxing conversions: int Foo<T>(T x) => (int)(object)x;Demikian pula untuk konversi unboxing: int Foo<T>(T x) => (int)(object)x;

## **Covariance and Contravariance (Variance)**Kovarian dan Kontravarian (Varian)

**Variance** is an advanced concept that allows for more flexible type compatibility with generic interfaces and delegates (and historically, arrays). It dictates when a generic type with one set of type arguments can be treated as the same generic type with different, but related, type arguments.Varian adalah konsep lanjutan yang memungkinkan kompatibilitas tipe yang lebih fleksibel dengan antarmuka generik dan delegasi (serta secara historis, array). Konsep ini menentukan kapan tipe generik dengan satu set argumen tipe dapat dianggap sebagai tipe generik yang sama dengan argumen tipe yang berbeda, tetapi terkait.

**Assumptions for Variance:** A is convertible to B (e.g., A subclasses B, or A implements B).Asumsi untuk Variasi: A dapat dikonversi menjadi B (misalnya, A merupakan subkelas dari B, atau A mengimplementasikan B).

### **Covariance (out modifier)**Kovarians (modifikator luar)

* **Definition:** If A is convertible to B, then GenericType<A> is convertible to GenericType<B>.Definisi: Jika A dapat dikonversi menjadi B, maka GenericType<A> dapat dikonversi menjadi GenericType<B>.
* **Modifier:** Achieved by marking a type parameter with the out modifier in an interface or delegate declaration (interface IPoppable<out T>).Modifier: Dicapai dengan menandai parameter tipe menggunakan modifier out dalam deklarasi antarmuka atau delegasi (antarmuka IPoppable<out T>).
* **Restriction:** The out type parameter can only appear in **output positions** (e.g., as a method return type, or a read-only property). It cannot be used as an input parameter for a method.Pembatasan: Parameter tipe keluaran hanya dapat muncul di posisi keluaran (misalnya, sebagai tipe kembalian metode, atau properti baca-saja). Parameter ini tidak dapat digunakan sebagai parameter masukan untuk sebuah metode.
* **Benefit:** Enables safe polymorphism. For example, if you have IPoppable<Bear>, you can assign it to IPoppable<Animal> because Bear "is an" Animal, and the interface only *outputs* T. You can't put an Animal (which might be a Camel) into a Stack<Bear>.Manfaat: Memungkinkan polimorfisme yang aman. Misalnya, jika Anda memiliki IPoppable<Bear>, Anda dapat menugaskan objek tersebut ke IPoppable<Animal> karena Bear “adalah” Animal, dan antarmuka hanya mengembalikan T. Anda tidak dapat memasukkan Animal (yang mungkin adalah Camel) ke dalam Stack<Bear>.

|  |
| --- |
| public interface IPoppable<out T> { T Pop(); } // T is covariant (output position)  // Example with Stack<T> implementing IPoppable<T> // var bears = new Stack<Bear>(); // IPoppable<Animal> animals = bears; // Legal! Can assign a stack of bears to a pop-able animal interface. // Animal a = animals.Pop(); // Safely pops an Animal (which will be a Bear) |

* Arrays historically support covariance (Bear[] can be implicitly converted to Animal[]), but this is **unsafe** for element assignments (animals[0] = new Camel() would cause a runtime error). Interface/delegate covariance is type-safe.Array secara historis mendukung kovarian (Bear[] dapat dikonversi secara implisit menjadi Animal[]), tetapi hal ini tidak aman untuk penugasan elemen (animals[0] = new Camel() akan menyebabkan kesalahan runtime). Kovarian antarmuka/delegasi aman secara tipe.

### **Contravariance (in modifier)**

* **Definition:** If A is convertible to B, then GenericType<B> is convertible to GenericType<A>. (The "reverse" conversion).Definisi: Jika A dapat dikonversi menjadi B, maka GenericType<B> dapat dikonversi menjadi GenericType<A>. (Konversi “balik”).
* **Modifier:** Achieved by marking a type parameter with the in modifier in an interface or delegate declaration (interface IPushable<in T>).Modifier: Dicapai dengan menandai parameter tipe menggunakan modifier in dalam deklarasi antarmuka atau delegasi (antarmuka IPushable<in T>).
* **Restriction:** The in type parameter can only appear in **input positions** (e.g., as a method parameter). It cannot be used as a return type or a read-only property.Pembatasan: Parameter tipe in hanya dapat muncul di posisi input (misalnya, sebagai parameter metode). Parameter ini tidak dapat digunakan sebagai tipe kembalian atau properti read-only.
* **Benefit:** Enables safe polymorphism where you can pass a more general type to something expecting a more specific type. For example, if you have IPushable<Animal>, you can assign it to IPushable<Bear> because anything that can push an Animal can certainly push a Bear.Manfaat: Memungkinkan polimorfisme yang aman, di mana Anda dapat meneruskan tipe yang lebih umum ke sesuatu yang mengharapkan tipe yang lebih spesifik. Misalnya, jika Anda memiliki IPushable<Animal>, Anda dapat menugaskan IPushable<Bear> karena apa pun yang dapat mendorong Animal tentu saja dapat mendorong Bear.

|  |
| --- |
| public interface IPushable<in T> { void Push(T obj); } // T is contravariant (input position)  IPushable<Animal> animals = new Stack<Animal>(); IPushable<Bear> bears = animals; // Legal! Assigning a push-able animal interface to a push-able bear interface. bears.Push(new Bear()); // Safely pushes a Bear (which is an Animal) |

* Common examples in .NET include IEnumerable<out T> (covariant) and IComparer<in T> (contravariant).Contoh umum dalam .NET meliputi IEnumerable<out T> (kovarian) dan IComparer<in T> (kontravarian).

**Why classes don't allow variance:** Concrete class implementations typically require data to flow in both directions (input and output), making it impossible to guarantee type safety with a single variance annotation. Interfaces and delegates, being abstract, can specify constraints on parameter usage.Mengapa kelas tidak mengizinkan varian: Implementasi kelas konkret umumnya memerlukan data untuk mengalir dalam dua arah (input dan output), sehingga tidak mungkin menjamin keamanan tipe dengan anotasi varian tunggal. Antarmuka dan delegasi, yang bersifat abstrak, dapat menentukan batasan penggunaan parameter.

In essence, C# generics offer a balance of flexibility, type safety, and efficient deployment by deferring type specialization to runtime, while C++ templates provide ultimate flexibility at the cost of compile-time code generation and source code distribution.Pada dasarnya, generik C# menawarkan keseimbangan antara fleksibilitas, keamanan tipe, dan implementasi yang efisien dengan menunda spesialisasi tipe hingga waktu eksekusi, sementara templat C++ memberikan fleksibilitas maksimal dengan biaya pembangkitan kode pada waktu kompilasi dan distribusi kode sumber.