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.

The primary benefits of generics, especially when compared to using `object` as a general type:

- **Increased Type Safety:** Errors are caught at compile time, not runtime.
- **Reduced Casting:** You avoid the need for explicit downcasting, making code cleaner and less error-prone.
- **Reduced Boxing/Unboxing:** For value types, generics eliminate the performance overhead associated with boxing and 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**.

Consider our `Stack` example, now made generic to stack instances of any type `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:

```
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.

You cannot instantiate an open generic type directly:

```
// 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).

Why Generics Exist

Generics address the challenges of writing reusable code that is both flexible and type-safe.

- **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.
- **Lack of Type Safety and Performance Issues (`object` type):** While using `object` (as in `ObjectStack` from earlier) allows for generality, it introduces problems:
  - **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.
  - **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.

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>`).

Generic Methods

A **generic method** declares type parameters within its own signature. This allows you to write methods that can operate on different types.

```
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:

```
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);`.

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.

```
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.

- A generic type or method can have **multiple type parameters**:

```
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.

Convention:

- For single type parameters, use `T`.
- For multiple type parameters, prefix with `T` and use descriptive names (e.g., `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.

```
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>`).

The `default` Generic Value

The `default` keyword is used to get the default value for a generic type parameter `T`.

- If `T` is a **reference type**, `default(T)` is `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`).

```
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;`.

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.

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:

```
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.

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."

Subclassing Generic Types

A generic class can be subclassed. The subclass can:

- **Leave the base class's type parameters open:**

  ```
  class Stack<T> { ... }
  class SpecialStack<T> : Stack<T> { ... }
  ```

- **Close the generic type parameters with a concrete type:**

  ```
  class IntStack : Stack<int> { ... }
  ```

Self-Referencing Generic Declarations

A type can refer to itself as the concrete type argument when closing a type parameter, typically used in interfaces like `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>`.

```
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.

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`.

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.

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

2. **Cast to `object` first:** Conversions to/from `object` are assumed not to be custom conversions. This resolves the ambiguity.

   ```
   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;`

Covariance and Contravariance (Variance)

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.

Assumptions for Variance: `A` is convertible to `B` (e.g., `A` subclasses `B`, or `A` implements `B`).

Covariance (`out` modifier)

- **Definition:** If `A` is convertible to `B`, then `GenericType<A>` is convertible to `GenericType<B>`.

- **Modifier:** Achieved by marking a type parameter with the `out` modifier in an interface or delegate declaration (`interface 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.

- **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>`.

```
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.

Contravariance (`in` modifier)

- **Definition:** If `A` is convertible to `B`, then `GenericType<B>` is convertible to `GenericType<A>`. (The "reverse" conversion).
- **Modifier:** Achieved by marking a type parameter with the `in` modifier in an interface or delegate declaration (`interface 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.
- **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`.

```
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).

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.

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.