**Delegates**Delegasi

A **delegate** is an object that knows how to call a method. Think of it as a blueprint for a method's signature. A delegate type defines:Delegasi adalah objek yang tahu cara memanggil metode. Anggap saja sebagai cetak biru untuk tanda tangan metode. Sebuah tipe delegasi mendefinisikan:

* The **return type** of the method it can reference.Tipe pengembalian dari metode yang dapat direferensikan.
* The **parameter types** of the method it can reference.Jenis parameter dari metode yang dapat direferensikan.

Once a delegate type is defined, you can create instances of that delegate, and each instance can "point" to a compatible method.Setelah tipe delegasi didefinisikan, Anda dapat membuat instance dari delegasi tersebut, dan setiap instance dapat "menunjuk" ke metode yang kompatibel.

**Defining and Using a Delegate**Mendefinisikan dan Menggunakan Delegasi

First, you **declare a delegate type**. This is similar to declaring a class or an interface, but it uses the delegate keyword:Pertama, Anda mendeklarasikan tipe delegasi. Hal ini mirip dengan mendeklarasikan kelas atau antarmuka, tetapi menggunakan kata kunci delegate:

|  |
| --- |
| delegate int Transformer(int x); // Defines a delegate type named TransformerMendefinisikan tipe delegasi bernama Transformer |

This declaration states that Transformer is compatible with any method that takes an int as input and returns an int.Deklarasi ini menyatakan bahwa Transformer kompatibel dengan metode apa pun yang mengambil sebuah int sebagai masukan dan mengembalikan sebuah int.

Next, you can have methods that match this signature:Selanjutnya, Anda dapat memiliki metode yang cocok dengan tanda tangan ini:

|  |
| --- |
| int Square(int x) { return x \* x; } // Or more concisely with an expression-bodied method: // int Square(int x) => x \* x; |

To use the delegate, you **assign a compatible method to a delegate variable**, which creates a delegate instance:Untuk menggunakan delegasi, Anda menetapkan metode yang kompatibel ke variabel delegasi, yang akan membuat instance delegasi:

|  |
| --- |
| Transformer t = Square; // 't' now "points" to the Square method |

This is shorthand for Transformer t = new Transformer(Square);. The Square here is a "method group" – the compiler picks the correct overload based on the delegate's signature.Ini adalah singkatan dari Transformer t = new Transformer(Square);. Square di sini adalah sebuah “kelompok metode” - kompiler memilih overload yang benar berdasarkan tanda tangan delegasi.

Finally, you can **invoke the delegate instance** just like a regular method:Terakhir, Anda dapat memanggil instance delegasi seperti metode biasa:

|  |
| --- |
| int answer = t(3); // Invokes the method currently referenced by 't' (Square(3)), result is 9 // This is shorthand for t.Invoke(3); |

**Complete Example:**

|  |
| --- |
| delegate int Transformer(int x); // Delegate type declaration  int Square(int x) => x \* x;  // ... in your main code: Transformer t = Square; // Create delegate instance int result = t(3); // Invoke delegate (calls Square(3)) Console.WriteLine(result); // Output: 9 |

The key idea is **indirection**. The caller invokes the delegate, and the delegate, in turn, calls its target method. This decouples the caller from the specific implementation of the target method.Delegasi sangat baik untuk membuat arsitektur plug-in atau mekanisme callback, di mana perilaku metode dapat dikustomisasi pada saat runtime.

**Writing Plug-in Methods with Delegates**

Delegates are excellent for creating **plug-in architectures** or **callback mechanisms**, where a method's behavior can be customized at runtime.

Consider a utility method Transform that applies an operation to each element in an array. Instead of hardcoding the operation, we can pass a delegate:Pertimbangkan metode utilitas Transform yang menerapkan sebuah operasi pada setiap elemen dalam larik. Alih-alih meng-coding operasi, kita bisa mengoper sebuah delegasi:

|  |
| --- |
| // Define the delegate type (as above) delegate int Transformer(int x);  int Square(int x) => x \* x; int Cube(int x) => x \* x \* x;  void Transform(int[] values, Transformer t) // 't' is a delegate parameter {  for (int i = 0; i < values.Length; i++)  values[i] = t(values[i]); // Invoke the plug-in method }  // ... in your main code: int[] values = { 1, 2, 3 };  Transform(values, Square); // Use Square method as the plug-in foreach (int i in values)  Console.Write(i + " "); // Output: 1 4 9  // Change the plug-in behavior easily: // Transform(values, Cube); // Would use Cube method instead |

Here, Transform is a **higher-order function** because it takes a function (represented by a delegate) as an argument.Di sini, Transform adalah fungsi tingkat tinggi karena mengambil sebuah fungsi (diwakili oleh sebuah delegasi) sebagai argumen.

**Instance and Static Method Targets**

A delegate can reference both static methods and instance methods.Delegasi dapat mereferensikan metode statis dan metode instance.

**Static Method Target:**

|  |
| --- |
| class Test { public static int Square(int x) => x \* x; } delegate int Transformer(int x);  // ... Transformer t = Test.Square; // Referencing a static method Console.WriteLine(t(10)); // Output: 100 |

**Instance Method Target:**

|  |
| --- |
| class Test { public int Square(int x) => x \* x; } delegate int Transformer(int x);  // ... Test test = new Test(); Transformer t = test.Square; // Referencing an instance method on a specific 'test' object Console.WriteLine(t(10)); // Output: 100 |

When an instance method is assigned to a delegate, the delegate object stores not only a reference to the method but also a reference to the specific **instance** of the object the method belongs to. This instance is accessible via the System.Delegate class's Target property (which is null for static methods). This ensures that the instance remains alive as long as the delegate is alive.Ketika sebuah metode instance ditugaskan ke sebuah delegasi, objek delegasi tidak hanya menyimpan referensi ke metode tapi juga referensi ke instance spesifik dari objek yang dimiliki oleh metode tersebut. Instance ini dapat diakses melalui properti Target kelas System.Delegate (yang bernilai nol untuk metode statis). Hal ini memastikan bahwa instance tetap hidup selama delegasi masih hidup.

**Multicast Delegates**

All delegate instances have **multicast capability**. This means a single delegate instance can reference **multiple target methods**. When the delegate is invoked, all referenced methods are called in the order they were added.Semua instance delegasi memiliki kemampuan multicast. Ini berarti satu instance delegasi dapat mereferensikan beberapa metode target. Ketika delegasi dipanggil, semua metode yang direferensikan akan dipanggil sesuai urutan penambahannya.

You combine delegate instances using the + and += operators:Anda menggabungkan instance delegasi menggunakan operator + dan +=:

|  |
| --- |
| delegate void SomeDelegate(); void SomeMethod1() { Console.WriteLine("Method 1"); } void SomeMethod2() { Console.WriteLine("Method 2"); }  SomeDelegate d = SomeMethod1; // d points to SomeMethod1 d += SomeMethod2; // d now points to both Method1 and Method2 // Invoking 'd' will call Method1, then Method2. |

The - and -= operators remove methods from a delegate's invocation list.Operator - dan -= menghapus metode dari daftar pemanggilan delegasi.

**Important:** Delegates are **immutable**. When you use += or -=, you are not modifying the existing delegate instance. Instead, a **new delegate instance** is created with the updated invocation list, and the variable is then assigned to this new instance.Penting: Delegasi tidak dapat diubah. Ketika Anda menggunakan += atau -=, Anda tidak memodifikasi instance delegasi yang sudah ada. Sebaliknya, instance delegasi baru dibuat dengan daftar pemanggilan yang diperbarui, dan variabel kemudian ditugaskan ke instance baru ini.

If a multicast delegate has a non-void return type, the caller receives the return value **only from the last method invoked**. Return values from preceding methods are discarded. For this reason, multicast delegates are most commonly used with void return types, particularly in event handling.Jika delegasi multicast memiliki tipe pengembalian non-void, pemanggil hanya menerima nilai pengembalian dari metode terakhir yang dipanggil. Nilai kembalian dari metode sebelumnya dibuang. Karena alasan ini, delegasi multicast paling sering digunakan dengan tipe pengembalian void, khususnya dalam penanganan event.

**Multicast Delegate Example (Progress Reporting):**

|  |
| --- |
| public delegate void ProgressReporter(int percentComplete);  public class Util {  public static void HardWork(ProgressReporter p)  {  for (int i = 0; i < 10; i++)  {  p(i \* 10); // Invoke the delegate to report progress  System.Threading.Thread.Sleep(100); // Simulate work  }  } }  // ... in your main code: void WriteProgressToConsole(int percentComplete) => Console.WriteLine($"Console: {percentComplete}%"); void WriteProgressToFile(int percentComplete) => File.WriteAllText("progress.txt", percentComplete.ToString());  ProgressReporter p = WriteProgressToConsole; // Start with console reporting p += WriteProgressToFile; // Add file reporting  Util.HardWork(p); // Both methods will be called as progress is made. |

**Generic Delegate Types**Jenis Delegasi Umum

Like classes and methods, delegate types can also have **generic type parameters**, allowing them to be highly reusable for various data types.Seperti halnya kelas dan metode, tipe delegasi juga dapat memiliki parameter tipe umum, yang memungkinkannya sangat dapat digunakan kembali untuk berbagai tipe data.

|  |
| --- |
| public delegate TResult Transformer<TArg, TResult>(TArg arg); // Generic delegate type // Example: Transformer<int, int> can refer to Square // Transformer<string, int> could refer to a method that counts characters in a string |

With this, you can create a truly generalized Transform method:Dengan ini, Anda dapat membuat metode Transform yang benar-benar umum:

|  |
| --- |
| public class Util {  public static void Transform<T>(T[] values, Transformer<T, T> t) // Uses generic delegate  {  for (int i = 0; i < values.Length; i++)  values[i] = t(values[i]);  } }  // ... int Square(int x) => x \* x; int[] values = { 1, 2, 3 }; Util.Transform(values, Square); // Works with ints |

**The Func and Action Delegates**Delegasi Fungsi dan Tindakan

C# provides a set of pre-defined generic delegates in the System namespace that cover almost all common method signatures. These are the Func and Action families:C# menyediakan satu set delegasi generik yang sudah ditentukan sebelumnya dalam ruang nama System yang mencakup hampir semua tanda tangan metode yang umum. Ini adalah keluarga Func dan Action:

* **Func<TResult>:** Represents a method that takes no arguments and returns a value of type TResult.Func<TResult>: Merepresentasikan metode yang tidak menerima argumen dan mengembalikan nilai tipe TResult.
* **Func<TArg, TResult>:** Represents a method that takes one argument of type TArg and returns a value of type TResult.Func<TArg, TResult>: Merepresentasikan metode yang mengambil satu argumen bertipe TArg dan mengembalikan nilai bertipe TResult.
* **Func<T1, T2, TResult>:** ... and so on, up to 16 input parameters.Func<T1, T2, TResult>: ... dan seterusnya, hingga 16 parameter input.
* **Action:** Represents a method that takes no arguments and returns void.Aksi: Merupakan metode yang tidak menerima argumen dan mengembalikan nilai kosong.
* **Action<TArg>:** Represents a method that takes one argument of type TArg and returns void.Action<TArg>: Merepresentasikan metode yang mengambil satu argumen bertipe TArg dan mengembalikan nilai void.
* **Action<T1, T2>:** ... and so on, up to 16 input parameters.Tindakan <T1, T2>: ... dan seterusnya, hingga 16 parameter input.

These delegates are incredibly versatile. Our Transformer<T, T> delegate above can be replaced directly with Func<T, T>:Delegasi ini sangat serbaguna. Delegasi Transformer<T, T> kami di atas dapat diganti secara langsung dengan Func<T, T>:

|  |
| --- |
| public static void Transform<T>(T[] values, Func<T, T> transformer) {  for (int i = 0; i < values.Length; i++)  values[i] = transformer(values[i]); } |

The only practical scenarios not covered by Func/Action are methods with ref or out parameters, or pointer parameters. Most new C# code prefers Func and Action over custom delegate types due to their generality.Satu-satunya skenario praktis yang tidak tercakup dalam Func/Action adalah metode dengan parameter ref atau out, atau parameter penunjuk. Sebagian besar kode C# baru lebih memilih Func dan Action daripada jenis delegasi khusus karena keumumannya.

**Delegates Versus InterfacesDelegasi Versus Antarmuka**

Many problems solvable with delegates can also be solved with interfaces. For example, our Transform method could use an interface:Banyak masalah yang dapat diselesaikan dengan delegasi juga dapat diselesaikan dengan antarmuka. Sebagai contoh, metode Transform kita dapat menggunakan antarmuka:

|  |
| --- |
| public interface ITransformer {  int Transform(int x); }  public class Util {  public static void TransformAll(int[] values, ITransformer t)  {  for (int i = 0; i < values.Length; i++)  values[i] = t.Transform(values[i]);  } }  class Squarer : ITransformer {  public int Transform(int x) => x \* x; }  // ... int[] values = { 1, 2, 3 }; Util.TransformAll(values, new Squarer()); // Uses an object implementing the interface |

**When to prefer a Delegate design:Kapan sebaiknya memilih desain Delegate:**

* **Single Method Interface:** If the interface would only define a single method. Delegates are more concise for this.Antarmuka Metode Tunggal: Jika antarmuka hanya akan mendefinisikan metode tunggal. Delegasi lebih ringkas untuk ini
* **Multicast Capability:** Delegates natively support multicasting; interfaces do not.Kemampuan Multicast: Delegasi secara bawaan mendukung multicasting; antarmuka tidak.
* **Multiple Implementations by Subscriber:** If a single class needs to provide *multiple different implementations* of the same "method signature" (e.g., Square and Cube transforms). With interfaces, you'd be forced to create separate classes (Squarer, Cuber) because a class can only implement an interface once. With delegates, a single class can expose multiple methods compatible with the delegate type.Beberapa Implementasi oleh Subscriber: Jika satu kelas perlu menyediakan beberapa implementasi yang berbeda dari “tanda tangan metode” yang sama (misal, transformasi Kotak dan Kubus). Dengan antarmuka, Anda akan dipaksa untuk membuat kelas terpisah (Squarer, Cuber) karena sebuah kelas hanya bisa mengimplementasikan sebuah antarmuka satu kali. Dengan delegasi, satu kelas dapat mengekspos beberapa metode yang kompatibel dengan tipe delegasi.

**Delegate Compatibility (Variance)**Kompatibilitas Delegasi (Varians)

Delegate types are considered **incompatible with one another** even if their signatures are identical, unless explicitly cast.Jenis delegasi dianggap tidak kompatibel satu sama lain meskipun tanda tangannya identik, kecuali secara eksplisit dilemparkan.

|  |
| --- |
| delegate void D1(); delegate void D2(); void Method1() { }  D1 d1 = Method1; // D2 d2 = d1; // Compile-time error: D1 and D2 are different types D2 d2 = new D2(d1); // OK: Can construct a new D2 from an existing D1 delegate instance |

Delegate instances are considered equal if they reference the same method targets in the same order (for multicast delegates).Instance delegasi dianggap sama jika mereferensikan target metode yang sama dengan urutan yang sama (untuk delegasi multicast). **Parameter Compatibility (Contravariance**)Kompatibilitas Parameter (Kontravarians)

A delegate's parameter types can be **more general** than the target method's parameter types. This is called **contravariance** (indicated by in for generic delegates).Tipe parameter delegasi bisa lebih umum daripada tipe parameter metode target. Ini disebut kontravarians (ditunjukkan dengan in untuk delegasi umum).

|  |
| --- |
| void ActOnObject(object o) => Console.WriteLine(o); delegate void StringAction(string s);  StringAction sa = new StringAction(ActOnObject); // Legal: string (more specific) can be passed to object (more general) sa("hello"); // "hello" is passed as string, then implicitly upcast to object by ActOnObject |

The StringAction expects a string, but ActOnObject takes an object. When sa("hello") is called, the string "hello" is provided. This string is then implicitly upcast to an object when it's passed to ActOnObject.StringAction mengharapkan sebuah string, namun ActOnObject mengambil sebuah objek. Ketika sa(“hello”) dipanggil, string “hello” disediakan. String ini kemudian secara implisit di-upcast ke sebuah objek ketika dioper ke ActOnObject.

**Return Type Compatibility (Covariance)Kompatibilitas Jenis Pengembalian (Kovarians)**

A delegate's return type can be **more general** than the target method's return type. This is called **covariance** (indicated by out for generic delegates).Tipe pengembalian delegasi bisa lebih umum daripada tipe pengembalian metode target. Ini disebut kovarians (ditunjukkan dengan out untuk delegasi umum).

|  |
| --- |
| string RetrieveString() => "hello"; delegate object ObjectRetriever();  ObjectRetriever o = new ObjectRetriever(RetrieveString); // Legal: object (more general) can receive string (more specific) object result = o(); // 'result' will be "hello" (a string, implicitly upcast to object) |

ObjectRetriever expects an object, but RetrieveString returns a string. The string return value is implicitly upcast to object when returned from the delegate.ObjectRetriever mengharapkan sebuah objek, tetapi RetrieveString mengembalikan sebuah string. Nilai pengembalian string secara implisit di-upcast ke objek ketika dikembalikan dari delegasi.

**Generic Delegate Type Parameter VarianceVarians Parameter Jenis Delegasi Generik**

Just like generic interfaces, generic delegates support in (contravariant) and out (covariant) type parameters. It's good practice to mark:Sama seperti antarmuka generik, delegasi generik mendukung parameter tipe masuk (kontravarian) dan keluar (kovarian). Merupakan praktik yang baik untuk menandai:

* **out** for type parameters used only in the return value.keluar untuk parameter tipe yang hanya digunakan dalam nilai balik.
* **in** for type parameters used only in parameters.in untuk parameter tipe yang hanya digunakan dalam parameter.

This allows for natural and type-safe conversions based on inheritance relationships, making your generic delegates more flexible. The Func and Action delegates in System namespace are designed with this variance in mind.Hal ini memungkinkan konversi alami dan aman berdasarkan hubungan pewarisan, sehingga delegasi generik Anda lebih fleksibel. Delegasi Func dan Action di ruang nama Sistem dirancang dengan mempertimbangkan perbedaan ini.