## **Equality Comparison**

In C#, determining if two objects are "equal" can mean different things depending on the context. Understanding these distinctions and the mechanisms for comparison is vital for correct program behavior, especially when working with custom types and collections.

### **Value Versus Referential Equality**

There are two fundamental kinds of equality:

1. **Value Equality:** Two values are considered equivalent if their *content* or *state* is the same. For example, two integers with the same numerical value are value-equal.
2. **Referential Equality:** Two references are considered equivalent if they point to *exactly the same object in memory*.

**Default Behavior:**

* **Value Types** (e.g., int, DateTimeOffset, structs): By default, value types use **value equality**. This means int x = 5; int y = 5; Console.WriteLine(x == y); will output True. Even with structs, by default, they exhibit **structural equality**, where all their members are compared for equality.
* **Reference Types** (e.g., class, string, Uri): By default, reference types use **referential equality**. This means two distinct objects, even if they have identical content, are not considered equal unless they are the *very same object*.

| class Foo { public int X; } Foo f1 = new Foo { X = 5 }; Foo f2 = new Foo { X = 5 }; Console.WriteLine(f1 == f2); // False (f1 and f2 point to different Foo objects)  Foo f3 = f1; Console.WriteLine(f1 == f3); // True (f3 points to the same object as f1) |
| --- |

* **Exceptions:** Some built-in reference types like string and System.Uri (and automatically generated anonymous types and records) *override* this default behavior to exhibit **value equality**.

| Uri uri1 = new Uri("http://www.linqpad.net"); Uri uri2 = new Uri("http://www.linqpad.net"); Console.WriteLine(uri1 == uri2); // True (Uri overrides == to use value equality) |
| --- |

### **Standard Equality Protocols**

C# and .NET provide three primary protocols for equality comparison:

1. **The == and != Operators:**
   * These are **static** operators, meaning the specific implementation used is determined at **compile time** based on the declared types of the operands. There's no virtual behavior.
   * For **value types**, == typically performs value equality.
   * For **reference types**, == typically performs referential equality (unless overloaded).
   * **Pitfall with object variables:** If you compare two boxed value types as object, the object type's == operator will be used, which performs referential equality, leading to False even if the underlying values are identical.

| object x = 5; // x boxes the integer 5 object y = 5; // y boxes another integer 5 Console.WriteLine(x == y); // False (x and y refer to different boxed objects) |
| --- |

1. **The Virtual Object.Equals Method:**
   * Equals is a **virtual** method defined in System.Object, meaning it's available to all types and its specific implementation is resolved at **runtime** based on the actual type of the object.
   * It typically performs **value equality** for value types (by default, structs compare all members) and **referential equality** for reference types (unless overridden).
   * Using Equals correctly equates x and y from the previous example:

| object x = 5; object y = 5; Console.WriteLine(x.Equals(y)); // True (Int32's Equals method is called, which compares values) |
| --- |

* + **Why == is not virtual/identical to Equals:**
    - Equals on a null object throws NullReferenceException, whereas == can handle null safely.
    - == operators are statically resolved and generally faster.
    - Sometimes, it's desirable for == and Equals to apply *different* definitions of equality (e.g., double.NaN).

1. **The Static Object.Equals Method:**
   * There's also a static Object.Equals(object objA, object objB) method. This is a helper that provides a **null-safe** comparison using the virtual Equals method if neither argument is null.
   * This is useful when you don't know the types at compile time or need to handle potential null values gracefully.

| object x = 3, y = 3; Console.WriteLine(object.Equals(x, y)); // True x = null; Console.WriteLine(object.Equals(x, y)); // False y = null; Console.WriteLine(object.Equals(x, y)); // True |
| --- |

* + This method is particularly helpful in generic code where the == operator might not be available or suitable due to type constraints.

1. **The Static Object.ReferenceEquals Method:**
   * This method *forces* a **referential equality** comparison, regardless of whether Equals or == have been overridden by the type.
   * It's useful when you explicitly need to check if two variables point to the exact same object in memory.

| Widget w1 = new Widget(); Widget w2 = new Widget(); Console.WriteLine(object.ReferenceEquals(w1, w2)); // False |
| --- |

1. **The IEquatable<T> Interface:**
   * public interface IEquatable<T> { bool Equals(T other); }
   * This interface provides a **type-specific** Equals method that avoids the boxing penalty for value types that would occur when calling Object.Equals.
   * It's recommended to implement IEquatable<T> when you override Object.Equals for performance.
   * You can use it as a generic constraint:

| class Test<T> where T : IEquatable<T> {  public bool IsEqual(T a, T b)  {  return a.Equals(b); // No boxing if T is a value type  } } |
| --- |

### 

### **When Equals and == Differ**

It is possible and sometimes necessary for Equals and == to behave differently:

* **double.NaN example:**

| double x = double.NaN; Console.WriteLine(x == x); // False (mathematically, NaN is never equal to anything, even itself) Console.WriteLine(x.Equals(x)); // True (Object.Equals must be reflexive: an object must equal itself) |
| --- |

* Collections and dictionaries rely on Equals being reflexive to correctly store and retrieve items.
* **StringBuilder example:** Some reference types, like StringBuilder, override Equals to perform value equality while leaving == to perform its default referential equality.

| var sb1 = new StringBuilder("foo"); var sb2 = new StringBuilder("foo"); Console.WriteLine(sb1 == sb2); // False (referential equality) Console.WriteLine(sb1.Equals(sb2)); // True (value equality, as implemented by StringBuilder) |
| --- |

### 

### **Equality and Custom Types: Customizing Logic**

You should consider customizing equality logic for your own types in two primary scenarios:

1. **To Change the Meaning of Equality:**
   * **For Classes:** If a class naturally represents a value (like string or Uri), it often makes sense to override Equals (and optionally ==) to perform value equality instead of the default referential equality.
   * **For Structs:** While structs default to structural value equality (comparing all fields), you might want to change this. For instance, in an Area struct, you might consider 5x10 and 10x5 to be equal, requiring custom logic. Records (since C# 9) and record structs (since C# 10) automatically implement structural equality by comparing all fields, but you can override this for specific fields or complex comparison needs (e.g., collections).
2. **To Speed Up Equality Comparisons for Structs:**
   * The default structural equality for structs is performed via reflection and can be slow. Overriding Equals can significantly improve performance. Overloading == and implementing IEquatable<T> allows for unboxed comparisons, further boosting performance.
   * *(Note: Customizing equality for reference types generally does not improve performance, as default referential equality is already very fast.)*

### **How to Override Equality Semantics**

For **classes and structs** (non-records), follow these steps:

1. **Override GetHashCode() and Equals(object other):**
   * **GetHashCode():** This method must return an int hash code for the object.
     + **Contract:** If a.Equals(b) is true, then a.GetHashCode() must equal b.GetHashCode(). (The reverse is not true.)
     + Hash codes should be as varied as possible to minimize "collisions" in hash-based collections (like Dictionary<TKey, TValue> and HashSet<T>), improving their performance.
     + The default GetHashCode for structs is often based on all fields; for classes, it's based on an internal object token (unique per instance).
     + Use HashCode.Combine() (from .NET Core 3.0 / .NET Standard 2.1 onwards) for easily combining multiple fields into a good hash code.
   * **Equals(object other):** This is the virtual method from Object. Your implementation should:
     + Handle null comparisons gracefully.
     + Check if other is of the correct type (other is MyType).
     + Call your type-specific Equals method (often the one from IEquatable<T>).
2. **(Optionally) Implement IEquatable<T>:**
   * public bool Equals(MyType other): This provides a type-safe and non-boxing Equals method for your type. Its logic should mirror Object.Equals.
3. **(Optionally) Overload == and != Operators:**
   * **Always overload both == and != together.**
   * For structs, this is almost always necessary, as without it, == and != simply won't work on your struct type.
   * For classes, you have a choice:
     + **Leave them alone:** They will perform default referential equality. This is common for mutable classes.
     + **Overload them to align with Equals (value equality):** This is typically done for immutable classes that conceptually represent a value (like string or Uri).

**For Records (and Record Structs) (C# 9+):**

The compiler automatically generates Equals, GetHashCode, ==, and != (and implements IEquatable<T>) based on **structural equality** (comparing all fields).

* If you need to customize this, you must override the generated Equals method with a specific signature (which is virtual and accepts your record type directly, not object).

| record Test (int X, int Y) {  public virtual bool Equals(Test? t) => t != null && t.X == X && t.Y == Y; } |
| --- |

* *(Note the virtual keyword instead of override for records' Equals.)*
* You **must also override GetHashCode()** if you customize Equals for records.
* You **do not need to** overload ==/!= or implement IEquatable<T> for records, as the compiler handles this for you based on your Equals implementation.

**Example: The Area Struct**

Consider an Area struct where 5x10 and 10x5 are considered equal.

| public struct Area : IEquatable<Area> {  public readonly int Measure1;  public readonly int Measure2;   public Area(int m1, int m2)  {  Measure1 = Math.Min(m1, m2); // Always store in a canonical order  Measure2 = Math.Max(m1, m2);  }   public override bool Equals(object? other) // Object.Equals override  => other is Area a && Equals(a); // Calls the IEquatable.Equals method   public bool Equals(Area other) // IEquatable<Area> implementation  => Measure1 == other.Measure1 && Measure2 == other.Measure2;   public override int GetHashCode() // GetHashCode override  => HashCode.Combine(Measure1, Measure2); // Efficiently combines hashes   public static bool operator ==(Area a1, Area a2) => Equals(a1, a2); // Overload ==  public static bool operator !=(Area a1, Area a2) => !(a1 == a2); // Overload != } |
| --- |