## **Dates and Times**

Dealing with dates and times correctly is crucial in many applications, from scheduling to logging. C# provides dedicated struct types in the System namespace to manage different aspects of temporal data. Importantly, these are **immutable structs**, meaning their values cannot change after they are created.

### **TimeSpan: An Interval of Time**

A TimeSpan represents a duration or an interval of time. It can also represent a "time of day" when treated as the elapsed time since midnight.

* **Resolution:** 100 nanoseconds (ns).
* **Range:** Approximately 10 million days, and can be positive or negative.

**Constructing a TimeSpan:**

1. **Constructors:** You can specify days, hours, minutes, seconds, milliseconds, and microseconds.

| public TimeSpan(int hours, int minutes, int seconds); public TimeSpan(int days, int hours, int minutes, int seconds, int milliseconds); public TimeSpan(long ticks); // Each tick = 100ns |
| --- |

1. **Static From... Methods:** Convenient for specifying an interval in a single unit.

| public static TimeSpan FromDays(double value); public static TimeSpan FromHours(double value); public static TimeSpan FromMinutes(double value); // ... and so on for Seconds, Milliseconds, Microseconds |
| --- |

1. **Example:**

| Console.WriteLine(new TimeSpan(2, 30, 0)); // 02:30:00 (2 hours, 30 minutes, 0 seconds) Console.WriteLine(TimeSpan.FromHours(2.5)); // 02:30:00 Console.WriteLine(TimeSpan.FromHours(-2.5)); // -02:30:00 |
| --- |

1. **Subtracting DateTime objects:** The difference between two DateTime or DateTimeOffset instances results in a TimeSpan.

**TimeSpan Operations:**

TimeSpan overloads operators like <, >, +, and - for intuitive calculations.

| TimeSpan duration = TimeSpan.FromHours(2) + TimeSpan.FromMinutes(30); // 02:30:00 TimeSpan nearlyTenDays = TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1); // 9.23:59:59 |
| --- |

**Properties:**

* **Integer Properties (Days, Hours, Minutes, Seconds, Milliseconds):** Return the component of the TimeSpan in their respective units (e.g., nearlyTenDays.Days would be 9).
* **Total... Properties (TotalDays, TotalHours, etc.):** Return the *entire* time span expressed as a double in the specified unit.

| Console.WriteLine(nearlyTenDays.TotalDays); // 9.999988... |
| --- |

**Conversion:**

* ToString(): Converts a TimeSpan to its string representation (e.g., "02:30:00").
* TimeSpan.Parse(): Converts a string to a TimeSpan.
* TimeSpan.TryParse(): Safely attempts conversion, returning false on failure instead of throwing an exception.
* The XmlConvert class also provides methods for TimeSpan/string conversion following XML protocols.

**Default Value: The default value for a TimeSpan is TimeSpan.Zero.**

TimeSpan can also represent the time of day; for instance, DateTime.Now.TimeOfDay returns a TimeSpan representing the current time since midnight.

### **DateTime and DateTimeOffset: Point in Time**

DateTime and DateTimeOffset are immutable structs representing a specific point in time (date and optionally time). They also have a 100ns resolution and a wide range (years 0001-9999).

Choosing Between DateTime and DateTimeOffset:

The primary difference lies in how they handle time zones:

* **DateTime**: Contains a three-state flag (DateTimeKind) indicating whether it's Local, Utc, or Unspecified.  
  + Local: Relative to the local time on the current computer.
  + Utc: Coordinated Universal Time (like GMT).
  + Unspecified: Time-zone agnostic (neither local nor UTC).
  + **Equality:** DateTime equality comparisons **ignore** the DateTimeKind flag. They consider two values equal if their year, month, day, hour, minute, etc., components match. This can be problematic when dealing with different time zones or Daylight Saving Time, as two DateTime values that represent the same *moment in time* might compare as unequal.
* **DateTimeOffset**: Stores the time along with an explicit **offset from UTC** as a TimeSpan (e.g., 2025-06-19 15:00:00 -07:00).  
  + **Equality:** DateTimeOffset equality comparisons consider two values equal if they refer to the **same point in absolute time**, regardless of their represented local time or offset.
  + **Advantage:** This is generally preferable for applications that deal with events across different time zones or need to store precise, unambiguous moments in time. It prevents issues related to Daylight Saving Time or different local times for the same event.
  + **Disadvantage:** DateTimeOffset does not store regional time zone information (e.g., "Pacific Standard Time"); it only stores the numeric offset from UTC.

**Recommendation:** For most scenarios involving past, present, or future events that need to be unambiguous across time zones (e.g., logging, database storage, scheduling international events), **DateTimeOffset is generally the safer and more robust choice**. DateTime is better suited for values relative to the local computer at runtime (e.g., "3 AM local time next Sunday" for scheduled tasks in different offices).

**Constructing a DateTime:**

* **Constructors:** Specify year, month, day, and optionally time components.

| public DateTime(int year, int month, int day); public DateTime(int year, int month, int day, int hour, int minute, int second, int millisecond); |
| --- |

* **DateTimeKind:** You can specify DateTimeKind.Unspecified (default), Local, or Utc in constructors.
* **Ticks:** From a long representing 100-ns intervals from 01/01/0001.
* **Static Parse/ParseExact:** To convert from string representations. ParseExact requires a specific format string.

**Constructing a DateTimeOffset:**

* **Constructors:** Similar to DateTime but you **must** also specify a TimeSpan for the UTC offset.

| public DateTimeOffset(int year, int month, int day, int hour, int minute, int second, TimeSpan offset); |
| --- |

* **From DateTime:** You can construct a DateTimeOffset from an existing DateTime. If no offset is specified, it's inferred from the DateTimeKind or the current local time zone.
  + DateTimeKind.Utc -> offset 0.
  + DateTimeKind.Local or Unspecified -> current local time zone offset.

**Converting Between DateTimeOffset and DateTime:**

DateTimeOffset provides properties to get DateTime values in specific forms:

* UtcDateTime: Returns DateTime in UTC time (Kind: Utc).
* LocalDateTime: Returns DateTime converted to the current local time zone (Kind: Local).
* DateTime: Returns DateTime with its Kind as Unspecified (effectively the UTC time plus its original offset).

**Current Date/Time:**

* DateTime.Now, DateTimeOffset.Now: Current local date and time.
* DateTime.Today: Current local date with time set to midnight.
* DateTime.UtcNow, DateTimeOffset.UtcNow: Current UTC date and time.

**Working with Dates and Times (Calculations):**

Both DateTime and DateTimeOffset provide instance methods for calculations:

* AddYears(), AddMonths(), AddDays(), AddHours(), AddMinutes(), AddSeconds(), AddMilliseconds(), AddTicks(): These methods return a *new* DateTime or DateTimeOffset object. They handle complexities like leap years automatically. You can pass negative values to subtract.
* Adding/Subtracting TimeSpan: The + and - operators are overloaded to add/subtract TimeSpan values.
* Subtracting DateTime/DateTimeOffset values: Returns a TimeSpan representing the duration between them.

**Formatting and Parsing:**

* ToString(): Converts DateTime or DateTimeOffset to a string representation, often influenced by the operating system's regional settings.
* ToShortDateString(), ToLongDateString(), ToShortTimeString(), ToLongTimeString(): Provide common predefined formats.
* ToString(string formatString, IFormatProvider provider): Allows extensive customization using **format strings** (e.g., "o" for round-trip format, "yyyy-MM-dd HH:mm:ss") and culture-specific formatting.
  + **Caution:** Relying solely on ToString() without a specific format string can lead to parsing issues if the culture settings differ between formatting and parsing. Using format strings like "o" (round-trip) is recommended for reliable conversions.
* Parse(), TryParse(), ParseExact(), TryParseExact(): Convert strings back into DateTime or DateTimeOffset objects. TryParse and TryParseExact are safer as they return false on failure instead of throwing an exception.

**Null DateTime/DateTimeOffset Values:**

Since these are structs (value types), they are not intrinsically nullable. To represent a "null" date/time:

* Use Nullable<DateTime> (i.e., DateTime?) or Nullable<DateTimeOffset> (i.e., DateTimeOffset?). This is generally the preferred and safest approach.
* Use DateTime.MinValue or DateTimeOffset.MinValue (the default value for these types). Be cautious with MinValue when performing conversions (e.g., ToUniversalTime()) as it might no longer be MinValue if crossing time zones.

### **DateOnly and TimeOnly (from .NET 6)**

Introduced in .NET 6, these structs provide more specific and type-safe ways to represent only a date or only a time, avoiding common pitfalls with DateTime.

* **DateOnly**: Represents just a date (year, month, day), without any time component.  
  + It lacks DateTimeKind and has no concept of Local or Utc.
  + **Advantage:** Prevents bugs that arise when a DateTime intended to be "just a date" accidentally acquires a non-zero time, causing equality comparisons to fail.
* **TimeOnly**: Represents just a time of day (hour, minute, second, nanosecond), without any date component.  
  + Ideal for scenarios like recording alarm times, opening hours, or recurring daily schedules.