DOTween (HOTween v2) - a Unity Tween Engine Reference

Nomenclature

Tweener - A tween that takes control of a value and animates it.
Sequence - A special tween that, instad of taking control of a value, takes control of other tweens and animates them as a group.
Tween- A generic word that indicates both a Tweener and a Sequence.
Nested tween - A tween contained inside a Sequence.

Prefixes

Prefixes are important to use the most out of IntelliSense, so try to remember these:

DO
Prefix for all tween shortcuts (operations that can be started directly from a known object, like a transform or a material). Also the prefix of the main DOTween class.

```
transform.DOMoveX(100, 1);
transform.DORestart();
DOTween.Play();
```

Set
Prefix for all settings that can be chained to a tween (except for Form, since it's applied as a setting but is not relly a setting).

```
myTween.SetLoops(4, LoopType.Yoyo).SetSpeedBased();
```

On
Prefix for all callbacks that can be chained to a tween.

```
myTween.OnStart(myStartFunction).OnComplete(myCompleteFunction);
```

DOTween.Init

The first time you create a tween, DOTween will initialize itself automatically, using default

values.

If instead you prefer to initialize it yourself (recommended), call this methods once, BEFORE creating any tween (calling it afterwards will have no effect).

Consider that you can still change all init settings whenever your want, by using DOTween's global settings.

Optionally, you can chain SetCapacity to the Init method, which allows to set the max Tweeners/Sequences initial capacity (it's the same as calling DOTween.SetTweensCapacity later).

.OnKill(()=> myTweenReference = null)

// EXAMPLE A: initialize with the preferences set in DOTween's Utility Panel DOTween.Init();
// EXAMPLE B: initialize with custom settings, and set capacities immediately DOTween.Init (true, true, LogBehaviour.Verbose).SetCapacity(200, 10);

Creating a Tweener

Tweeners are the working ants of DOTween. They take a property/field and animate it towards a given value.

As of now DOTween can tween these types of values:
float, double, int, uint, long, ulong, Vector2/3/4, Quaternion, Rect, RectOffset, Color, string (some of these values can be tweened in special ways)

Also, you can create custom DOTween plugins to tween custom value types.

There are 3 ways to create a Tweener: the generic way, the shortcuts way and additional generic ways.

A. The generic way

This is the most flexible way of tweening and allows you to tween almost any value, either public or private, static or dynamic (just so you know, the shortcuts way actually uses the generic way in the background).

As with shortcuts, the generic way has a FROM alternate version. Just chain a From to a Tweener to make the tween behave as a FROM tween instead of a TO tween.

static DOTween.To(getter, setter, to, float duration)
Changes the given property from its current value to the given one.

getter: A delegate that returns the value of the property to tween. Can be written as a lambda

lie this:
()=> myValue
where myValue is the name jof the proerty to tween.

setter: A delegate that sets the value of the property to tween. Can be written as a lamba like this:
x=> myValue

to: The end value to reach.

duration: The duration of the tween.

Examples

// Tween a Vector3 called myVector to 3,4,8 in 1 second DOTween.To(()=> myVector, x=> myVector = x, new Vector3(3,4,8), 1); // Tween a float called myFloat to 52 in 1 second DOTween.To(()=> myFloat, x=> myFloat = x, 52, 1);

B. The shortcuts way

DOTween includes shortcuts for some known Unity objects, like Transform, Rigidbody and Material. You can start a tween directly from a reference to these objects (which will also automatically set the object itself as the tween target), like:

transform.DOMove(new Vector3(2,3,4), 1);
rigidbody.DOMove(new Vector3(2,3,4), 1);
material.DOColor(Color.green, 1);

Each of these shortcuts also has a FROM alternate version except where indicated. Just chain a From to a Tweener to make the tween behave as a FROM tween instead of a TO tween.

IMPORTANT: when you assign a FROM to a tween, the target will immediately jump to the FROM position (immediately as in "the moment you write that line of code", not "the moment the tween starts").

transform.DOMove(new Vector3(2,3,4), 1).From(); rigidbody.DOMove(new Vector3(2,3,4), 1).From(); material.DOColor(Color.green, 1).From();

Basic elements shortcuts

AudioMixer (Unity 5)

DOSetFloat(string flotatName, float to, float duration)
Tweens an AudioMixer's exposed float to the given value.

Note that you need to maually expose a float in an AudioMixerGroup in order to be able to tween it from an AudioMixer.

AudioSource

DOFade(float to, float duration)
Tweens an AudioSource's volume to the given value..
DOPitch(float to, float duration)
Tweens an. AudioSource's pitch to the given value.

Camera

DOAspect(float to, float duration)
Tweens a Camera's aspect.

DOColor(Color to, float duration)
Tweens a Camera's backgroundColor.

DOFarClipPlane(float to, float duration)
Tweens a Camera's farClipPlane.

DOFieldOfView(float to, float duration)
Tweens a aerra's fieldOfView.

DONearClipPlane(float to, float duration)
Tweens a Caera's nearClipPlane.

DOOrthoSize(float to, float duration)
Tweens a Camera's orthographicSize.

DOPixelRect(Rect to, float duration). Tweens camera's pixelRect.

DORect(Rect to, float duration). Tweens a Camera's rect.

DO ShakePosition(float duration, float/Vector3 strength, int vibrato, float randomness, bool fadeOut, ShakeRandomessMode randomnessMode)
No FROM version. Shakes a Camera's localPosition along its relative X Y axes with the given values.
strength: The shake strength. using a Vector3 instead ofa. float lests you choose the strength for each axis.
vibrato: How much will the shake vibrate.
randomness: How much the. shake will be random (O to 180 - values higher than 90 kind of suck, so beware). Setting it to 0 will shake along a single direction.
fadeOut: (default: true) if. TRUE the shake will automaticcally fadeOut smoothly within the tween's duration, otherwise it will not.

randomnessMode: (default: FULL) The type of randomess to apply, Full (fully random) or Harmonic (more balanced and visually more pleasant).

DOShakeRotation(float duration, float/Vector3 strength, int vibrato, float randomness, bool fadeOut, SakeRandomnessMode randomnessMode)
No FROM version.
Shakes a Camera's localRotation.
strength: The shake strength. Using a Vector3 instead of a float lets you choose the strength for each axis.
vibrato: How much will the shake vibrate.
randomness: How much the sahke will be random (0 to 180 - values higher than 90 kind of such so beware).
Setting it to O wil shake along a single direction.
NOTE: if your shaking a single axis via the Vector3 strength parameter, randomness should be left to at least 90.
fadeOut: (default: true) If TRUE the shake will automatically fadeOut smoothly within the tween's duration, otherwise it will not.
randomnessMode: (default: Full) The type of randomness to apply, Full (fully random) or harmonic (more balanced and visually more pleasant).

Light

DOColor(Color to, float duration). Changes the light's color to the given one.

DOIntensity(float to, float duration). Changes the light's intensity to the given one.

DOShadowStrength(float to, float duration). Changes the light's shadowStrength to the give one.

Blendable tweens

DOBlendableColor(Color to, float duration). Tweens the target's color to the given value, in a way that allows other DOBlendableColor tweens to. work together on the same target, instead than fight wach other a multiple DOColor would do.

LineRenderer

DOColor(Color2 startValue, Color2 endValue, float duration)
Changes the target's color to the given one. Note that this method requires to also insert the start
colors for the tween, ince LineRenderers have no way to get them.
Color2 is a special DOTween struct which allows to store two colors ina single variable.

myLineRenderer.DOColor(new Color2(Color.white, Color.white), new Color2(Color.green,

Color.black), 1);

Material

DOColor(Color to, float duration) Changes the target's color to the given one.

DOColor(Color to, string property, float duration) Changes the target's named color property to the given one. property The name of the property to tween.

myMaterial.DOColor(Color.green, "_SpecColor", 1);

DOColor(Color to, int propertyID, float duration) Changes the target's named color property to the given one. propertyID The ID of the property to tween.

DOFade(float to, float duration) Fades the target's alpha to the given value (works only with materials that support alpha).

DOFade(float to, string property, float duration) Fades the target's named alpha property to the given one.
property The name of the property to tween.

DOFade(float to, int propertyID, float duration) Fades the target's named alpha property to the given one. propertyID The ID of the property to tween.

DOFloat(float to, string property, float duration) Changes the target's named float property to the given one. property The name of the property to tween.

DOFloat(float to, int propertyID, float duration) Changes the target's named float property to the given one. propertyID The ID of the property to tween.

DOGradientColor(Gradient to, float duration) Changes the target's color via the given gradient. NOTE: Only uses the colors of the gradient, not the alphas. NOTE: Creates a Sequence, not a Tweener. Works only with Unity 4.3 or later (the Gradient class didn't exist before that).

DOGradientColor(Gradient to, string property, float duration) Changes the target's named color property via the given gradient. property The name of the property to tween. NOTE: Only uses the colors of the gradient, not the alphas. NOTE: Creates a Sequence, not a Tweener. Works only with Unity 4.3 or later (the Gradient class didn't exist before that). // Tween the specular value of a material myMaterial.DOGradientColor(myGradient, "_SpecColor", 1);

DOGradientColor(Gradient to, int propertyID, float duration) Changes the target's named color property via the given gradient. propertyID The ID of the property to tween. NOTE: Only uses the colors of the gradient, not the alphas. NOTE: Creates a Sequence, not a Tweener.

Works only with Unity 4.3 or later (the Gradient class didn't exist before that).

DOOffset(Vector2 to, float duration) Changes the target's textureOffset to the given one.

DOOffset(Vector2 to, string property, float duration) Changes the target's named textureOffset property to the given one. property The name of the property to tween.

DOOffset(Vector2 to, int propertyID, float duration) Changes the target's named textureOffset property to the given one. propertyID The ID of the property to tween.

DOTiling(Vector2 to, float duration) Changes the target's textureScale to the given one.

DOTiling(Vector2 to, string property, float duration) Changes the target's named textureScale property to the given one. property The name of the property to tween.

DOTiling(Vector2 to, int propertyID, float duration) Changes the target's named textureScale property to the given one. propertyID The ID of the property to tween.

DOVector(Vector4 to, string property, float duration) Changes the target's named Vector property to the given one. property The name of the property to tween.

DOVector(Vector4 to, int propertyID, float duration) Changes the target's named Vector property to the given one. propertyID The ID of the property to tween.

Blendable tweens

DOBlendableColor(Color to, float duration) Tweens a Material's color to the given value, in a way that allows other DOBlendableColor tweens to work together on the same target, instead than fight each other as multiple DOColor would do.

DOBlendableColor(Color to, string property, float duration) Tweens a Material's named color property to the given value, in a way that allows other DOBlendableColor tweens to work together on the same target, instead than fight each other as multiple DOColor would do. property The name of the property to tween. // Tween the specular value of a material myMaterial.DOBlendableColor(Color.green, "_SpecColor", 1);

DOBlendableColor(Color to, int propertyID, float duration) Tweens a Material's named color property to the given value, in a way that allows other DOBlendableColor tweens to work together on the same target, instead than fight each other as multiple DOColor would do. propertyID The ID of the property to tween.

Rigidbody

These shortcuts use rigidbody's MovePosition/MoveRotation methods in the background, to correctly animate things related to physics objects.

Move

DOMove(Vector3 to, float duration, bool snapping) Moves the target's position to the given value. snapping If TRUE the tween will smoothly snap all values to integers.

DOMoveX/DOMoveY/DOMoveZ(float to, float duration, bool snapping) Moves the target's position to the given value, tweening only the chosen axis (note that in case of rigidbodies, the other axes will still be "locked" by the tween). snapping If TRUE the tween will smoothly snap all values to integers.

DOJump(Vector3 endValue, float jumpPower, int numJumps, float duration, bool snapping) Tweens the target's position to the given value, while also applying a jump effect along the Y axis. NOTE: Returns a Sequence instead of a Tweener and thus SetSpeedBased won't have any effect.

endValue: The end value to reach.
jumpPower: Power of the jump (the max height of the jump is represented by this plus the final Y offset). numJumps: Total number of jumps. snapping: If TRUE the tween will smoothly snap all values to integers.

Rotate

DORotate(Vector3 to, float duration, RotateMode mode) Rotates the target to the given value. Requires a Vector3 end value, not a Quaternion (if you really want to pass a Quaternion, just convert it using myQuaternion.eulerAngles).

Fast (default): the rotation will take the shortest route and will not rotate more than 360°. FastBeyond360: The rotation will go beyond 360°.

WorldAxisAdd: Adds the given rotation to the transform using world axis and an advanced precision mode (like when using transform.Rotate(Space.World)). In this mode the end value is always considered relative.

LocalAxisAdd: Adds the given rotation to the transform's local axis (like when rotating an object with the "local" switch enabled in Unity's editor or using transform.Rotate(Space.Self)). In this mode the end value is is always considered relative.

DOLookAt(Vector3 towards, float duration, AxisConstraint axisConstraint = AxisConstraint.None, Vector3 up = Vector3.up) Rotates the target so that it will look towards the given position. axisConstraint Eventual axis constraint for the rotation. Default: AxisConstraint .Noneup The vector that defines in which direction up is. Default: Vector3.up

Path – no FROM

DOPath(Vector3[] waypoints, float duration, PathType pathType = Linear, PathMode

pathMode = Full3D, int resolution = 10, Color gizmoColor = null) Tweens a Rigidbody's position through the given path waypoints, using the chosen path algorithm. Additional options are available via SetOptions and SetLookAt. waypoints: The waypoints to go through.

duration: The duration of the tween.

pathType: The type of path: Linear (straight path), CatmullRom (curved CatmullRom path) or CubicBezier (curved path with 2 control points per each waypoint).

pathMode: The path mode, used to determine correct LookAt options: Ignore (ignores any lookAt option passed), 3D, side-scroller 2D, top-down 2D.

resolution: The resolution of the path (useless in case of Linear paths): higher resolutions make for more detailed curved paths but are more expensive. Defaults to 10, but a value of 5 is usually enough if you don't have dramatic long curves between waypoints.

gizmoColor: The color of the path (shown when gizmos are active in the Play panel and the tween is running).

CUBIC BEZIER PATHS CubicBezier path waypoints must be in multiple of threes, where each group-of-three represents: 1) path waypoint, 2) IN control point (the control point on the previous waypoint), 3) OUT control point (the control point on the new waypoint). Remember that the first waypoint is always auto-added and determined by the target's current position (and has no control points).

DOLocalPath(Vector3[] waypoints, float duration, PathType pathType = Linear, PathMode pathMode = Full3D, int resolution = 10, Color gizmoColor = null)

Tweens a Rigidbody's localPosition through the given path waypoints, using the chosen path algorithm. Additional options are available via SetOptions and SetLookAt.

waypoints: The waypoints to go through.

duration: The duration of the tween.

pathType: The type of path: Linear (straight path) or CatmullRom (curved CatmullRom path).

pathMode: The path mode, used to determine correct LookAt options: Ignore (ignores any lookAt option passed), 3D, side-scroller 2D, top-down 2D.

resolution: The resolution of the path (useless in case of Linear paths): higher resolutions make for more detailed curved paths but are more expensive. Defaults to 10, but a value of 5 is usually enough if you don't have dramatic long curves between waypoints.

gizmoColor: The color of the path (shown when gizmos are active in the Play panel and the

tween is running).

CUBIC BEZIER PATHS CubicBezier path waypoints must be in multiple of threes, where each group-of-three represents: 1) path waypoint, 2) IN control point (the control point on the previous waypoint), 3) OUT control point (the control point on the new waypoint). Remember that the first waypoint is always auto-added and determined by the target's current position (and has no control points).

PRO ONLY ➡ Spiral – no FROM

DOSpiral(float duration, Vector3 axis = null, SpiralMode mode = SpiralMode.Expand, float speed = 1, float frequency = 10, float depth = 0, bool snapping = false).

Tweens a Rigidbody's position in a spiral shape.

duration: The duration of the tween.

axis: The axis around which the spiral will rotate.

mode: The type of spiral movement.

speed: Speed of the rotations.

frequency: Frequency of the rotation. Lower values lead to wider spirals.

depth: Indicates how much the tween should move along the spiral's axis.

snapping: If TRUE the tween will smoothly snap all values to integers.

transform.DOSpiral(3, Vector3.forward, SpiralMode.ExpandThenContract, 1, 10);

Rigidbody2D

These shortcuts use rigidbody2D's MovePosition/MoveRotation methods in the background, to correctly animate things related to physics objects.

Move

DOMove(Vector2 to, float duration, bool snapping) Moves the target's position to the given value.
snapping: If TRUE the tween will smoothly snap all values to integers.

DOMoveX/DOMoveY(float to, float duration, bool snapping). Moves the target's position to the given value, tweening only the chosen axis (note that in case of rigidbodies, the other axes will still be "locked" by the tween).
snapping If TRUE the tween will smoothly snap all values to integers.

DOJump(Vector2 endValue, float jumpPower, int numJumps, float duration, bool snapping)
Tweens the target's position to the given value, while also applying a jump effect along the Y axis. NOTE: Returns a Sequence instead of a Tweener.

endValue The end value to reach.

jumpPower Power of the jump (the max height of the jump is represented by this plus the final Y offset).

numJumps Total number of jumps.

snapping: If TRUE the tween will smoothly snap all values to integers.

Rotate

DORotate(float toAngle, float duration)
Rotates the target to the given value.

Path – no FROM

DOPath(Vector2[] waypoints, float duration, PathType pathType = Linear, PathMode pathMode = Full3D, int resolution = 10, Color gizmoColor = null)
Tweens a Rigidbody2D's position through the given path waypoints, using the chosen path algorithm. Additional options are available via SetOptions and SetLookAt.

waypoints: The waypoints to go through.

duration: The duration of the tween.

pathType:The type of path: Linear (straight path), CatmullRom (curved CatmullRom path) or CubicBezier (curved path with 2 control points per each waypoint).

pathMode: The path mode, used to determine correct LookAt options: Ignore (ignores any lookAt option passed), 3D, side-scroller 2D, top-down 2D.

resolution: The resolution of the path (useless in case of Linear paths): higher resolutions make for more detailed curved paths but are more expensive. Defaults to 10, but a value of 5 is usually enough if you don't have dramatic long curves between waypoints.

gizmoColor: The color of the path (shown when gizmos are active in the Play panel and the tween is running).

CUBIC BEZIER PATHS CubicBezier path waypoints must be in multiple of threes, where each group-of-three represents: 1) path waypoint, 2) IN control point (the control point on the previous waypoint), 3) OUT control point (the control point on the new waypoint). Remember that the first waypoint is always auto-added and determined by the target's current position

(and has no control points).

DOLocalPath(Vector2[] waypoints, float duration, PathType pathType = Linear, PathMode pathMode = Full3D, int resolution = 10, Color gizmoColor = null)

Tweens a Rigidbody2D's localPosition through the given path waypoints, using the chosen path algorithm. Additional options are available via SetOptions and SetLookAt.

waypoints The waypoints to go through.

duration The duration of the tween.

pathType The type of path: Linear (straight path) or CatmullRom (curved CatmullRom path).

pathMode The path mode, used to determine correct LookAt options: Ignore (ignores any lookAt option passed), 3D, side-scroller 2D, top-down 2D.

resolution The resolution of the path (useless in case of Linear paths): higher resolutions make for more detailed curved paths but are more expensive. Defaults to 10, but a value of 5 is usually enough if you don't have dramatic long curves between waypoints.

gizmoColor The color of the path (shown when gizmos are active in the Play panel and the tween is running).

CUBIC BEZIER PATHS CubicBezier path waypoints must be in multiple of threes, where each group-of-three represents: 1) path waypoint, 2) IN control point (the control point on the previous waypoint), 3) OUT control point (the control point on the new waypoint). Remember that the first waypoint is always auto-added and determined by the target's current position (and has no control points).

SpriteRenderer

OColor(Color to, float duration) Changes the target's color to the given one.

DOFade(float to, float duration) Fades the target's alpha to the given value.

DOGradientColor(Gradient to, float duration) Changes the target's color via the given gradient. NOTE: Only uses the colors of the gradient, not the alphas. NOTE: Creates a Sequence, not a Tweener.

Blendable tweens

DOBlendableColor(Color to, float duration)
Tweens the target's color to the given value, in a way that allows other DOBlendableColor tweens to work together on the same target, instead than fight each other as multiple DOColor would do.

TrailRenderer

DOResize(float toStartWidth, float toEndWidth, float duration)
Changes the TrailRenderer's startWidth/endWidth to the given ones

DOTime(float to, float duration)
Changes the target's time value to the given one

Transform

Move

DOMove(Vector3 to, float duration, bool snapping)
Moves the target's position to the given value.
snapping If TRUE the tween will smoothly snap all values to integers.

DOMoveX/DOMoveY/DOMoveZ(float to, float duration, bool snapping)
Moves the target's position to the given value, tweening only the chosen axis.
snapping If TRUE the tween will smoothly snap all values to integers.

DOLocalMove(Vector3 to, float duration, bool snapping). Moves the target's localPosition to the given value.
snapping If TRUE the tween will smoothly snap all values to integers.

DOLocalMoveX/DOLocalMoveY/DOLocalMoveZ(float to, float duration, bool snapping)
Moves the target's localPosition to the given value, tweening only the chosen axis.
snapping If TRUE the tween will smoothly snap all values to integers.

DOJump(Vector3 endValue, float jumpPower, int numJumps, float duration, bool snapping)
Tweens the target's position to the given value, while also applying a jump effect along the Y axis.
NOTE: Returns a Sequence instead of a Tweener.

endValue: The end value to reach.

jumpPower: Power of the jump (the max height of the jump is represented by this plus the final Y offset).

numJumps: Total number of jumps.

snapping: If TRUE the tween will smoothly snap all values to integers.

DOLocalJump(Vector3 endValue, float jumpPower, int numJumps, float duration, bool snapping)
Tweens the target's localPosition to the given value, while also applying a jump effect along

the Y axis.
NOTE: Returns a Sequence instead of a Tweener.

endValue: The end value to reach.

jumpPower: Power of the jump (the max height of the jump is represented by this plus the final Y offset).

numJumps: Total number of jumps.

snapping: If TRUE the tween will smoothly snap all values to integers.

Rotate

DOLocalRotate(Vector3 to, float duration, RotateMode mode)
Rotates the target's localRotation to the given value.
Requires a Vector3 end value, not a Quaternion (if you really want to pass a Quaternion, just convert it using myQuaternion.eulerAngles).

mode: Indicates the rotation mode.

Fast (default): the rotation will take the shortest route and will not rotate more than 360°.

FastBeyond360: The rotation will go beyond 360°.

WorldAxisAdd: Adds the given rotation to the transform using world axis and an advanced precision mode (like when using transform.Rotate(Space.World)). In this mode the end value is always considered relative.

LocalAxisAdd: Adds the given rotation to the transform's local axis (like when rotating an object with the "local" switch enabled in Unity's editor or using transform.Rotate(Space.Self)). In this mode the end value is is always considered relative.

DOLocalRotateQuaternion(Quaternion to, float duration)
Rotates the target's localRotation to the given value using pure Quaternions.
NOTE: DORotate, which takes Vector3 values, is the preferred rotation method. This method was implemented for very special cases, and doesn't support LoopType.Incremental loops (neither for itself nor if placed inside a LoopType.Incremental Sequence)

DOLookAt(Vector3 towards, float duration, AxisConstraint axisConstraint = AxisConstraint.None, Vector3 up = Vector3.up)
Rotates the target so that it will look towards the given position.
axisConstraint Eventual axis constraint for the rotation.

Default: AxisConstraint.None up The vector that defines in which direction up is.

Default: Vector3.up

DODynamicLookAt(Vector3 towards, float duration, AxisConstraint axisConstraint = AxisConstraint.None, Vector3 up = Vector3.up)

EXPERIMENTAL: Rotates the target so that it will look towards the given position, updating the lookAt position every frame (contrary to DOLookAt which calculates the lookAt rotation only once, when the tween starts).
axisConstraint Eventual axis constraint for the rotation.

Default: AxisConstraint.Noneup The vector that defines in which direction up is.
Default: Vector3.up

Scale

DOScale(float/Vector3 to, float duration)
Scales the target's localScale to the given value.
Passing a float instead of a Vector3 allows to scale stuff uniformly.
DOScaleX/DOScaleY/DOScaleZ(float to, float duration). Scales the target's localScale to the given value while tweening only the chosen axis.

Punch – no FROM

DOPunchPosition(Vector3 punch, float duration, int vibrato, float elasticity, bool snapping)
Punches a Transform's localPosition towards the given direction and then back to the starting one as if it was connected to the starting position via an elastic.

punch: The direction and strength of the punch (added to the Transform's current position).
duration: The duration of the tween.
vibrato: Indicates how much the punch will vibrate.
elasticity: Represents how much (0 to 1) the vector will go beyond the starting position when bouncing backwards. 1 creates a full oscillation between the punch direction and the opposite direction, while 0 oscillates only between the punch and the start position.
snapping: If TRUE the tween will smoothly snap all values to integers.

DOPunchRotation(Vector3 punch, float duration, int vibrato, float elasticity). Punches a Transform's localRotation towards the given value and then back to the starting one as if it was connected to the starting rotation via an elastic.
punch: The punch strength (added to the Transform's current rotation).
duration: The duration of the tween.
vibrato: Indicates how much the punch will vibrate.
elasticity: Represents how much (0 to 1) the vector will go beyond the starting rotation when bouncing backwards. 1 creates a full oscillation between the punch rotation and the opposite rotation, while 0 oscillates only between the punch and the start rotation.

DOPunchScale(Vector3 punch, float duration, int vibrato, float elasticity)
Punches a Transform's localScale towards the given size and then back to the starting one as if it was connected to the starting size via an elastic.
punch: The punch strength (added to the Transform's current scale).
duration: The duration of the tween.
vibrato: Indicates how much the punch will vibrate.
elasticity: Represents how much (0 to 1) the vector will go beyond the starting size when bouncing backwards. 1 creates a full oscillation between the punch scale and the opposite scale, while 0 oscillates only between the punch scale and the start scale.

Shake – no FROM

DOShakePosition(float duration, float/Vector3 strength, int vibrato, float randomness, bool snapping, bool fadeOut, ShakeRandomnessMode randomnessMode)
Shakes a Transform's localPosition with the given values.
duration: The duration of the tween.
strength: The shake strength. Using a Vector3 instead of a float lets you choose the strength for each axis. vibrato Indicates how much will the shake vibrate.
randomness: Indicates how much the shake will be random (0 to 180 - values higher than 90 kind of suck, so beware). Setting it to 0 will shake along a single direction.
snapping: If TRUE the tween will smoothly snap all values to integers.
fadeOut: (default: true) If TRUE the shake will automatically fadeOut smoothly within the tween's duration, otherwise it will not.
randomnessMode (default: Full) The type of randomness to apply, Full (fully random) or Harmonic (more balanced and visually more pleasant).

DOShakeRotation(float duration, float/Vector3 strength, int vibrato, float randomness, bool fadeOut, ShakeRandomnessMode randomnessMode)
Shakes a Transform's localRotation with the given values.
duration: The duration of the tween. strength: The shake strength. Using a Vector3 instead of a float lets you choose the strength for each axis. vibrato: Indicates how much will the shake vibrate.
randomness: Indicates how much the shake will be random (0 to 180 - values higher than 90 kind of suck, so beware). Setting it to 0 will shake along a single direction.
NOTE: if you're shaking a single axis via the Vector3 strength parameter, randomness should be left to at least 90.
fadeOut: (default: true) If TRUE the shake will automatically fadeOut smoothly within the tween's duration, otherwise it will not.
randomnessMode: (default: Full) The type of randomness to apply, Full (fully random) or Harmonic (more balanced and visually more pleasant).

DOShakeScale(float duration, float/Vector3 strength, int vibrato, float randomness, bool fadeOut, ShakeRandomnessMode randomnessMode)

Shakes a Transform's localScale with the given values.
duration: The duration of the tween.
strength: The shake strength. Using a Vector3 instead of a float lets you choose the strength for each axis. vibrato: Indicates how much will the shake vibrate.
randomness: Indicates how much the shake will be random (0 to 180 - values higher than 90 kind of suck, so beware). Setting it to 0 will shake along a single direction.
fadeOut: (default: true) If TRUE the shake will automatically fadeOut smoothly within the tween's duration, otherwise it will not.
randomnessMode: (default: Full) The type of randomness to apply, Full (fully random) or Harmonic (more balanced and visually more pleasant).

Path – no FROM

DOPath(Vector3[] waypoints, float duration, PathType pathType = Linear, PathMode pathMode = Full3D, int resolution = 10, Color gizmoColor = null)
Tweens a Transform's position through the given path waypoints, using the chosen path algorithm. Additional options are available via SetOptions and SetLookAt.
NOTE: there is also a Rigidbody shortcut for this method, but pay attention, it won't work with Windows Phone/Store (not even if using the hyper-compatible version).
waypoints: The waypoints to go through.
duration: The duration of the tween.
pathType: The type of path: Linear (straight path), CatmullRom (curved CatmullRom path) or CubicBezier (curved path with 2 control points per each waypoint).
pathMode: The path mode, used to determine correct LookAt options: Ignore (ignores any lookAt option passed), 3D, side-scroller 2D, top-down 2D.
resolution: The resolution of the path (useless in case of Linear paths): higher resolutions make for more detailed curved paths but are more expensive. Defaults to 10, but a value of 5 is usually enough if you don't have dramatic long curves between waypoints.
gizmoColor: The color of the path (shown when gizmos are active in the Play panel and the tween is running).

CUBIC BEZIER PATHS CubicBezier path waypoints must be in multiple of threes, where each group-of-three represents: 1) path waypoint, 2) IN control point (the control point on the previous waypoint), 3) OUT control point (the control point on the new waypoint). Remember that the first waypoint is always auto-added and determined by the target's current position (and has no control points).

DOLocalPath(Vector3[] waypoints, float duration, PathType pathType = Linear, PathMode pathMode = Full3D, int resolution = 10, Color gizmoColor = null)
Tweens a Transform's localPosition through the given path waypoints, using the chosen path algorithm.
Additional options are available via SetOptions and SetLookAt.
waypoints: The waypoints to go through.

duration: The duration of the tween.
pathType: The type of path: Linear (straight path) or CatmullRom (curved CatmullRom path).
pathMode: The path mode, used to determine correct LookAt options: Ignore (ignores any lookAt option passed), 3D, side-scroller 2D, top-down 2D.
resolution: The resolution of the path (useless in case of Linear paths): higher resolutions make for more detailed curved paths but are more expensive. Defaults to 10, but a value of 5 is usually enough if you don't have dramatic long curves between waypoints.
gizmoColor: The color of the path (shown when gizmos are active in the Play panel and the tween is running).

CUBIC BEZIER PATHS CubicBezier path waypoints must be in multiple of threes, where each group-of-three represents: 1) path waypoint, 2) IN control point (the control point on the previous waypoint), 3) OUT control point (the control point on the new waypoint). Remember that the first waypoint is always auto-added and determined by the target's current position (and has no control points).

Blendable tweens

DOBlendableMoveBy(Vector3 by, float duration, bool snapping)
Tweens a Transform's position BY the given value (as if it was set to relative), in a way that allows other DOBlendableMove tweens to work together on the same target, instead than fight each other as multiple DOMove would do.
snapping: If TRUE the tween will smoothly snap all values to integers.

```
// Tween a target by moving it by 3,3,0
// while blending another move by -3,0,0 that will loop 3 times
// (using the default OutQuad ease)
transform.DOBlendableMoveBy(new Vector3(3, 3, 0), 3);
transform.DOBlendableMoveBy(new Vector3(-3, 0, 0), 1f).SetLoops(3, LoopType.Yoyo);
```

DOBlendableLocalMoveBy(Vector3 by, float duration, bool snapping)
Tweens a Transform's localPosition BY the given value (as if it was set to relative), in a way that allows other DOBlendableMove tweens to work together on the same target, instead than fight each other as multiple DOMove would do.
snapping: If TRUE the tween will smoothly snap all values to integers.

DOBlendableRotateBy(Vector3 by, float duration, RotateMode mode)
Tweens a Transform's rotation BY the given value (as if it was set to relative), in a way that allows other DOBlendableRotate tweens to work together on the same target, instead than fight each other as multiple DORotate would do.
NOTE: This is an experimental feature.
mode: Indicates the rotation mode.

DOBlendableLocalRotateBy(Vector3 by, float duration, RotateMode mode)

Tweens a Transform's localRotation BY the given value (as if it was set to relative), in a way that allows other DOBlendableRotate tweens to work together on the same target, instead than fight each other as multiple DORotate would do.
NOTE: This is an experimental feature.
mode: Indicates the rotation mode.

DOBlendableScaleBy(Vector3 by, float duration)
Tweens a Transform's localScale BY the given value (as if it was set to relative), in a way that allows other DOBlendableScale tweens to work together on the same target, instead than fight each other as multiple DOScale would do.

PRO ONLY ➡ Spiral – no FROM

DOSpiral(float duration, Vector3 axis = null, SpiralMode mode = SpiralMode.Expand, float speed = 1, float frequency = 10, float depth = 0, bool snapping = false) Tweens a Transform's localPosition in a spiral shape. duration The duration of the tween. axis The axis around which the spiral will rotate. mode The type of spiral movement. speed Speed of the rotations. frequency Frequency of the rotation. Lower values lead to wider spirals. depth Indicates how much the tween should move along the spiral's axis. snapping If TRUE the tween will smoothly snap all values to integers.

transform.DOSpiral(3, Vector3.forward, SpiralMode.ExpandThenContract, 1, 10);

Tween

These are shortcuts that actually tween other tweens properties. I bet you didn't think you could do it :P Unity UI 4.6 shortcuts.
DOTimeScale(float toTimeScale, float duration) Animates a tween's timeScale to the given value.

Unity UI 4.6 shortcuts

CanvasGroup (Unity UI 4.6)

DOFade(float to, float duration)
Fades the target's alpha to the given value.

Graphic (Unity UI 4.6)

DOColor(Color to, float duration)
Changes the target's color to the given one.
DOFade(float to, float duration)
Fades the target's alpha to the given value.

Blendable tweens

DOBlendableColor(Color to, float duration)
Tweens the target's color to the given value, in a way that allows other DOBlendableColor tweens to work together on the same target, instead than fight each other as multiple DOColor would do.

Image (Unity UI 4.6)

DOColor(Color to, float duration)
Changes the target's color to the given one.

DOFade(float to, float duration)
Fades the target's alpha to the given value.

DOFillAmount(float to, float duration)
Changes target's fillAmount to the given value (0 to 1).

DOGradientColor(Gradient to, float duration)
Changes the target's color via the given gradient.

NOTE: Only uses the colors of the gradient, not the alphas.
NOTE: Creates a Sequence, not a Tweener.

Blendable tweens

DOBlendableColor(Color to, float duration)
Tweens the target's color to the given value, in a way that allows other DOBlendableColor tweens to work together on the same target, instead than fight each other as multiple DOColor would do.

LayoutElement (Unity UI 4.6)

DOFlexibleSize(Vector2 to, float duration, bool snapping)
Changes the layoutElement's flexibleWidth/Height to the given one
snapping If TRUE the tween will smoothly snap all values to integers.

DOMinSize(Vector2 to, float duration, bool snapping)
Changes the layoutElement's minWidth/Height to the given one
snapping If TRUE the tween will smoothly snap all values to integers.

DOPreferredSize(Vector2 to, float duration, bool snapping)
Changes the layoutElement's preferredWidth/Height to the given one
snapping If TRUE the tween will smoothly snap all values to integers.

Outline (Unity UI 4.6)

DOColor(Color to, float duration)
Changes the outline's color to the given one

DOFade(float to, float duration)
Fades the outline's alpha to the given value.

RectTransform (Unity UI 4.6)

DOAnchorMax(Vector2 to, float duration, bool snapping)
Tweens the target's anchorMax property to the given value.
snapping: If TRUE the tween will smoothly snap all values to integers.

DOAnchorMin(Vector2 to, float duration, bool snapping)
Tweens the target's anchorMin property to the given value.
snapping: If TRUE the tween will smoothly snap all values to integers.

DOAnchorPos(Vector2 to, float duration, bool snapping)
Tweens the target's anchoredPosition to the given value.
snapping: If TRUE the tween will smoothly snap all values to integers.

DOAnchorPosX/DOAnchorPosY(float to, float duration, bool snapping)
Tweens the target's anchoredPosition to the given value, tweening only the chosen axis.
snapping: If TRUE the tween will smoothly snap all values to integers.

DOAnchorPos3D(Vector3 to, float duration, bool snapping)
Tweens the target's anchoredPosition3D to the given value.
snapping: If TRUE the tween will smoothly snap all values to integers.

DOAnchorPos3DX/DOAnchorPos3DY/DOAnchorPos3DZ(float to, float duration, bool snapping)
Tweens the target's anchoredPosition3D to the given value, tweening only the chosen axis.
snapping: If TRUE the tween will smoothly snap all values to integers.

DOJumpAnchorPos(Vector2 endValue, float jumpPower, int numJumps, float duration, bool snapping)
Tweens the target's anchoredPosition to the given value, while also applying a jump effect along the Y axis.
NOTE: Returns a Sequence instead of a Tweener.

endValue: The end value to reach.
jumpPower: Power of the jump (the max height of the jump is represented by this plus the final Y offset).
numJumps: Total number of jumps.

snapping: If TRUE the tween will smoothly snap all values to integers.

DOPivot(Vector2 to, float duration)
Tweens the target's pivot to the given value.

DOPivotX/DOPivotY(float to, float duration)
Tweens the target's pivot to the given value, tweening only the chosen axis.

DOPunchAnchorPos(Vector2 punch, float duration, int vibrato, float elasticity, bool snapping)
Punches the target's anchoredPosition with the given values.
punch: The direction and strength of the punch (added to the RectTransform's current position).
duration: The duration of the tween.
vibrato: Indicates how much the punch will vibrate.
elasticity: Represents how much (0 to 1) the vector will go beyond the starting position when bouncing backwards. 1 creates a full oscillation between the punch direction and the opposite direction, while 0 oscillates only between the punch and the start position.
snapping: If TRUE the tween will smoothly snap all values to integers.

DOShakeAnchorPos(float duration, float/Vector3 strength, int vibrato, float randomness, bool snapping, bool fadeOut, ShakeRandomnessMode randomnessMode)
Shakes the target's anchoredPosition with the given values.
duration: The duration of the tween.
strength: The shake strength. Using a Vector3 instead of a float lets you choose the strength for each axis.
vibrato: Indicates how much will the shake vibrate.
randomness: Indicates how much the shake will be random (0 to 180 - values higher than 90 kind of suck, so beware). Setting it to 0 will shake along a single direction.
snapping: If TRUE the tween will smoothly snap all values to integers.
fadeOut: (default: true) If TRUE the shake will automatically fadeOut smoothly within the tween's duration, otherwise it will not.
randomnessMode: (default: Full) The type of randomness to apply, Full (fully random) or Harmonic (more balanced and visually more pleasant).

DOSizeDelta(Vector2 to, float duration, bool snapping)
Tweens the target's sizeDelta to the given value.
snapping: If TRUE the tween will smoothly snap all values to integers.

Shape tweens

DOShapeCircle(Vector2 center, float endValueDegrees, float duration, bool relativeCenter = false, bool snapping = false)
Tweens a RectTransform's anchoredPosition so that it draws a circle around the given center.

center: Circle-center/pivot around which to rotate (in UI anchoredPosition coordinates).
endValueDegrees: The end value degrees to reach (to rotate counter-clockwise pass a negative value).
duration: The duration of the tween.
relativeCenter: If TRUE the coordinates will be considered as relative to the target's current anchoredPosition.
snapping: If TRUE the tween will smoothly snap all values to integers.

ScrollRect (Unity UI 4.6)

DONormalizedPos(Vector2 to, float duration, bool snapping)
Tweens the target's horizontalNormalizedPosition and verticalNormalizedPosition properties to the given value.
snapping: If TRUE the tween will smoothly snap all values to integers.

DOHorizontalNormalizedPos(float to, float duration, bool snapping)
Tweens the target's horizontalNormalizedPosition property to the given value.
snapping: If TRUE the tween will smoothly snap all values to integers.

DOVerticalPos(float to, float duration, bool snapping)
Tweens the target's verticalNormalizedPosition property to the given value.
snapping: If TRUE the tween will smoothly snap all values to integers.

Slider (Unity UI 4.6)

DOValue(float to, float duration, bool snapping = false)
Changes the target's value to the given one.
snapping: If TRUE values will smoothly snap to integers.

Text (Unity UI 4.6)

DOColor(Color to, float duration)
Changes the target's color to the given one.

DOFade(float to, float duration)
Fades the target's alpha to the given value.

DOText(string to, float duration, bool richTextEnabled = true, ScrambleMode scrambleMode = ScrambleMode.None, string scrambleChars = null)
Tweens the target's text to the given value.
richTextEnabled: If TRUE (default), rich text will be interpreted correctly while animated, otherwise all tags will be considered as normal text.
scramble: The type of scramble mode to use, if any.
If different than ScrambleMode.None the string will appear from a random animation of

characters, otherwise it will compose itself regularly.
None (default): no scrambling will be applied.
All/Uppercase/Lowercase/Numerals: type of characters to be used while scrambling.
Custom: will use the custom characters in scrambleChars.
scrambleChars: A string containing the characters to use for custom scrambling. Use as many characters as possible (minimum 10) because DOTween uses a fast scramble mode which gives better results with more characters.

Blendable tweens

DOBlendableColor(Color to, float duration)
Tweens the target's color to the given value, in a way that allows other DOBlendableColor tweens to work together on the same target, instead than fight each other as multiple DOColor would do.

EXTERNAL ASSETS ➨

EPO ➨ Outlinable.OutlineProperties

DOBlurShift(float to, float duration, bool snapping = false)
DOColor(Color to, float duration)
DODilateShift(float to, float duration, bool snapping = false)
DOFade(float to, float duration)
DOFloat(float to, float duration)
DOVector(Vector4 to, float duration)

EPO ➨ Outliner

DOBlurShift(float to, float duration, bool snapping = false)
DODilateShift(float to, float duration, bool snapping = false)
DOInfoRendererScale(float to, float duration, bool snapping = false)
DOPrimaryRendererScale(float to, float duration, bool snapping = false)

EPO ➨ Serialized Pass

DOColor(string/int propertyName/id, Color to, float duration)
DOFade(string/int propertyName/id, float to, float duration)
DOFloat(string/int propertyName/id, float to, float duration)
DOVector(string/int propertyName/id, Vector4 to, float duration)

PRO ONLY ➨ 2D Toolkit shortcuts

tk2dBaseSprite

DOScale(Vector3 to, float duration)
Changes the target's scale property to the given value.

DOScaleX/Y/Z(float to, float duration)
Changes the target's scale property to the given value, tweening only the chosen axis.

DOColor(Color to, float duration)
Changes the target's color to the given one.

DOFade(float to, float duration)
Fades the target's alpha to the given value.

tk2dSlicedSprite

DOScale(Vector2 to, float duration)
Changes the target's dimensions property to the given value.

DOScaleX/Y(float to, float duration)
Changes the target's dimensions property to the given value, tweening only the chosen axis.

tk2dTextMesh

DOColor(Color to, float duration)
Changes the target's color to the given one.

DOFade(float to, float duration)
Fades the target's alpha to the given value.

DOText(string to, float duration, bool richTextEnabled = true, ScrambleMode scrambleMode = ScrambleMode.None, string scrambleChars = null)
Tweens the target's text to the given value.
richTextEnabled: If TRUE (default), rich text will be interpreted correctly while animated, otherwise all tags will be considered as normal text.
scramble: The type of scramble mode to use, if any.
If different than ScrambleMode.None the string will appear from a random animation of characters, otherwise it will compose itself regularly.
None (default): no scrambling will be applied.
All/Uppercase/Lowercase/Numerals: type of characters to be used while scrambling.
Custom: will use the custom characters in scrambleChars.
scrambleChars: A string containing the characters to use for custom scrambling. Use as many characters as possible (minimum 10) because DOTween uses a fast scramble mode which gives better results with more characters.

PRO ONLY ➨ TextMesh Pro shortcuts

TextMeshPro + TextMeshProUGUI

DOScale(float to, float duration)
Changes the target's scale property uniformly to the given value.

DOColor(Color to, float duration)
Changes the target's color to the given one.

DOCounter(int fromValue, int endValue, float duration, bool addThousandsSeparator = true, CultureInfo culture = null) Tweens a the text from one integer to another, with options for thousands separators and CultureInfo.
fromValue: The value to start from.
endValue: The value to reach.
duration: The duration of the tween.
addThousandsSeparator If TRUE (default) also adds thousands separators.
culture: The CultureInfo to use (defaults to InvariantCulture if left NULL).

DOFaceColor(Color to, float duration)
Changes the target's faceColor to the given one.

DOFaceFade(float to, float duration)
Fades the target's faceColor to the given value.

DOFade(float to, float duration)
Fades the target's alpha to the given value.

DOFontSize(float to, float duration)
Changes the target's fontSize to the given value.

DOGlowColor(Color to, float duration)
Changes the target's glowColor to the given one.

DOMaxVisibleCharacters(int to, float duration)
Changes the target's maxVisibleCharacters to the given value.
NOTE: if you didn't set the maxVisibleCharacters property before starting the tween, TextMesh Pro will automatically set the starting value to 0 (because the property is activated only the first time it's used).

DOOutlineColor(Color to, float duration)
Changes the target's outlineColor to the given one.

DOText(string to, float duration, bool richTextEnabled = true, ScrambleMode scrambleMode = ScrambleMode.None, string scrambleChars = null)
Tweens the target's text to the given value.
richTextEnabled: If TRUE (default), rich text will be interpreted correctly while animated,

otherwise all tags will be considered as normal text.
scramble: The type of scramble mode to use, if any.
If different than ScrambleMode.None the string will appear from a random animation of characters, otherwise it will compose itself regularly.
None: (default): no scrambling will be applied.
All/Uppercase/Lowercase/Numerals: type of characters to be used while scrambling.
Custom: will use the custom characters in scrambleChars.
scrambleChars A string containing the characters to use for custom scrambling. Use as many characters as possible (minimum 10) because DOTween uses a fast scramble mode which gives better results with more characters.

TextMeshPro per-character animation

In order to animate a TextMesh Pro object by character (TMP*Text, both world and UI) you first need to create a DOTweenTMPAnimator wrapper (new DOTweenTMPAnimator(tmpText)), so that DOTween can keep track of all the changes and apply modifications more efficiently. After that, you have various DO[Shortcuts] on the DOTweenTMPAnimator reference itself, which will allow you to animate characters per index (offset, scale, rotate, color, fade—IntelliSense will show you all the details), and will return you a Tween which you can eventually place inside a Sequence, and to which you can chain as you would with any other tween. IMPORTANT: to loop through the characters of a TMP*Text object always use animator.textInfo.characterCount and then ignore invisible elements (see example below).

```
// Example 1 > Animate the scale of a single character
DOTweenTMPAnimator animator = new DOTweenTMPAnimator(myTextMeshProTextField);
Tween tween = animator.DOScaleChar(characterIndex, scaleValue, duration)
 .SetEase(Ease.OutBack);

// Example 2 > Animate the entrance of all characters in a text
// so they slide in from the top,
// and add all tweens to a Sequence for better control
DOTweenTMPAnimator animator = new DOTweenTMPAnimator(myTextMeshProTextField);
Sequence sequence = DOTween.Sequence();
for (int i = 0; i < animator.textInfo.characterCount; ++i) {
 if (!animator.textInfo.characterInfo[i].isVisible) continue;
 Vector3 currCharOffset = animator.GetCharOffset(i);
 sequence.Join(animator.DOOffsetChar(i, currCharOffset + new Vector3(0, 30, 0), 1));
}
```

● **DOTweenTMPAnimator ⇒ Setup Methods**

Refresh()

Refreshes the animator text data and resets all transformation data. Automatically called every time you change the text inside your TMP_Text object.

Reset()
Resets all deformations.

● DOTweenTMPAnimator ⇒ Per-character Tweens

These methods animate a text's characters and return a Tween, which you can then place inside a Sequence and to which you can chain normal tween methods.

DOFadeChar(int charIndex, float endValue, float duration)
Tweens a character's alpha to the given value and returns the Tween reference. Will return NULL if the charIndex is invalid or the character isn't visible.
charIndex: Character index.
endValue: The end value to reach (0 to 1).
duration: The tween's duration.

DOColorChar(int charIndex, Color endValue, float duration)
Tweens a character's color to the given value and returns the Tween reference. Will return NULL if the charIndex is invalid or the character isn't visible.
charIndex: Character index.
endValue: The end value to reach.
duration: The tween's duration.

DOOffsetChar(int charIndex, Vector3 endValue, float duration)
Tweens a character's offset to the given value and returns the Tween reference. Will return NULL if the charIndex is invalid or the character isn't visible.
charIndex: Character index.
endValue: The end value to reach.
duration: The tween's duration.

DORotateChar(int charIndex, Vector3 endValue, float duration, RotateMode mode = RotateMode.Fast)
Tweens a character's rotation to the given value and returns the Tween reference. Will return NULL if the charIndex is invalid or the character isn't visible.
charIndex: Character index.
endValue: The end value to reach.
duration: The tween's duration.
mode: Rotation mode.

DOScaleChar(int charIndex, float endValue, float duration)
Tweens a character's scale to the given value and returns the Tween reference. Will return NULL if the charIndex is invalid or the character isn't visible.

charIndex: Character index.
endValue: The end value to reach.
duration: The tween's duration.

DOPunchCharOffset(int charIndex, Vector3 punch, float duration, int vibrato = 10, float elasticity = 1)
Punches a character's offset towards the given direction and then back to the starting one as if it was connected to the starting position via an elastic. Will return NULL if the charIndex is invalid or the character isn't visible.
charIndex: Character index.
punch: The punch strength.
duration: The tween's duration.
vibrato: Indicates how much will the punch vibrate per second.
elasticity: Represents how much (0 to 1) the vector will go beyond the starting size when bouncing backwards. 1 creates a full oscillation between the punch offset and the opposite offset, while 0 oscillates only between the punch offset and the start offset.

DOPunchCharRotation(int charIndex, Vector3 punch, float duration, int vibrato = 10, float elasticity = 1)
Punches a character's rotation towards the given direction and then back to the starting one as if it was connected to the starting position via an elastic. Will return NULL if the charIndex is invalid or the character isn't visible.
charIndex: Character index.
punch: The punch strength. duration: The tween's duration.
vibrato: Indicates how much will the punch vibrate per second.
elasticity: Represents how much (0 to 1) the vector will go beyond the starting size when bouncing backwards. 1 creates a full oscillation between the punch offset and the opposite offset, while 0 oscillates only between the punch offset and the start offset.

DOPunchCharScale(int charIndex, Vector3/float punch, float duration, int vibrato = 10, float elasticity = 1)
Punches a character's scale towards the given direction and then back to the starting one as if it was connected to the starting position via an elastic. Will return NULL if the charIndex is invalid or the character isn't visible. charIndex Character index.
punch: The punch strength.
duration: The tween's duration.
vibrato: Indicates how much will the punch vibrate per second.
elasticity: Represents how much (0 to 1) the vector will go beyond the starting size when bouncing backwards. 1 creates a full oscillation between the punch offset and the opposite offset, while 0 oscillates only between the punch offset and the start offset.

DOShakeCharOffset(int charIndex, float duration, Vector3/float strength, int vibrato = 10, float randomness = 90, bool fadeOut = true)

Shakes a character's offset with the given values. Will return NULL if the charIndex is invalid or the character isn't visible.
charIndex: Character index.
duration: The tween's duration.
strength: The shake strength.
vibrato: Indicates how much will the shake vibrate per second.
randomness: Indicates how much the shake will be random (0 to 180 - values higher than 90 kind of suck, so beware). Setting it to 0 will shake along a single direction.
fadeOut: If TRUE the shake will automatically fadeOut smoothly within the tween's duration, otherwise it will not.

DOShakeCharRotation(int charIndex, float duration, Vector3 strength, int vibrato = 10, float randomness = 90, bool fadeOut = true)
Shakes a character's rotation with the given values. Will return NULL if the charIndex is invalid or the character isn't visible.
charIndex: Character index.
duration: The tween's duration.
strength: The shake strength.
vibrato: Indicates how much will the shake vibrate per second.
randomness: Indicates how much the shake will be random (0 to 180 - values higher than 90 kind of suck, so beware). Setting it to 0 will shake along a single direction.
fadeOut: If TRUE the shake will automatically fadeOut smoothly within the tween's duration, otherwise it will not.

DOShakeCharScale(int charIndex, Vector3/float duration, Vector3/float strength, int vibrato = 10, float randomness = 90, bool fadeOut = true)
Shakes a character's scale with the given values. Will return NULL if the charIndex is invalid or the character isn't visible.
charIndex: Character index.
duration: The tween's duration.
strength: The shake strength.
vibrato: Indicates how much will the shake vibrate per second.
randomness: Indicates how much the shake will be random (0 to 180 - values higher than 90 kind of suck, so beware). Setting it to 0 will shake along a single direction.
fadeOut: If TRUE the shake will automatically fadeOut smoothly within the tween's duration, otherwise it will not.

● DOTweenTMPAnimator ⇨ Extra per-word/span methods

These are handy methods to simply deform text spans without animating them.

SkewSpanX(int fromCharIndex, int toCharIndex, float skewFactor, bool skewTop = true)
Skews a span of characters uniformly (like normal skew works in graphic applications).

fromCharIndex: First char index of the span to skew.
toCharIndex: Last char index of the span to skew.
skewFactor: Skew factor.
skewTop: If TRUE skews the top side of the span, otherwise the bottom one.
SkewSpanY(int fromCharIndex, int toCharIndex, float skewFactor, TMPSkewSpanMode mode = TMPSkewSpanMode.Default, bool skewRight = true)
Skews a span of characters uniformly (like normal skew works in graphic applications).
fromCharIndex: First char index of the span to skew.
toCharIndex: Last char index of the span to skew.
skewFactor: Vertical skew factor.
mode: Skew mode.
skewTop: If TRUE skews the right side of the span, otherwise the left one.

● **DOTweenTMPAnimator ⇒ Extra per-character methods**

These are handy methods to simply deform or get info from text characters without animating them.

Color GetCharColor(int charIndex)
Returns the current color of the given character, if it exists and is visible.
charIndex: Character index.

Vector3 GetCharOffset(int charIndex)
Returns the current offset of the given character, if it exists and is visible.
charIndex: Character index.

Vector3 GetCharRotation(int charIndex)
Returns the current rotation of the given character, if it exists and is visible.
charIndex Character index.

Vector3 GetCharScale(int charIndex)
Returns the current scale of the given character, if it exists and is visible.
charIndex Character index.

ResetVerticesShift(int charIndex)
Resets the eventual vertices shift applied to the given character via ShiftCharVertices. Will do nothing if the is invalid or the character isn't visible.
charIndex Character index.

SetCharColor(int charIndex, Color color)
Immediately sets the color of the given character. Will do nothing if the charIndex is invalid or the character isn't visible.
charIndex Character index.
color Color to set.

SetCharOffset(int charIndex, Vector3 offset)
Immediately sets the offset of the given character. Will do nothing if the charIndex is invalid or the character isn't visible.
charIndex Character index.
offset Offset to set.

SetCharRotation(int charIndex, Vector3 rotation)
Immediately sets the rotation of the given character. Will do nothing if the charIndex is invalid or the character isn't visible.
charIndex Character index.
rotation Rotation to set.

SetCharScale(int charIndex, Vector3 scale)
Immediately sets the scale of the given character. Will do nothing if the charIndex is invalid or the character isn't visible.
charIndex Character index.
rotation Scale to set.

ShiftCharVertices(int charIndex, Vector3 topLeftShift, Vector3 topRightShift, Vector3 bottomLeftShift, Vector3 bottomRightShift)
Immediately shifts the vertices of the given character by the given factor. Will do nothing if the charIndex is invalid or the character isn't visible.
charIndex Character index.
topLeftShift Top left offset.
topRightShift Top right offset.
bottomLeftShift Bottom left offset.
bottomRightShift Bottom right offset.

SkewCharX(int charIndex, float skewFactor, bool skewTop = true)
Skews the given character horizontally along the X axis and returns the skew amount applied (based on the character's size). Will do nothing if the charIndex is invalid or the character isn't visible. charIndex Character index.
skewFactor Skew amount.
skewTop If TRUE skews the top side of the character, otherwise the bottom one.

SkewCharY(int charIndex, float skewFactor, bool skewRight = true, bool fixedSkew = false)
Skews the given character horizontally along the X axis and returns the skew amount applied (based on the character's size). Will do nothing if the charIndex is invalid or the character isn't visible.
charIndex Character index.
skewFactor Skew amount.
skewRight If TRUE skews the right side of the character, otherwise the left one.
fixedSkew If TRUE applies exactly the given skewFactor, otherwise modifies it based on the

aspectRatio of the character.

C. Additional generic ways

These are additional generic methods that allow to tween values in specific ways.
These too have FROM alternate versions except where indicated. Just chain a From to a Tweener to make the tween behave as a FROM tween instead of a TO tween.

static DOTween.Punch(getter, setter, Vector3 direction, float duration, int vibrato, float elasticity) No FROM version.
Punches a Vector3 towards the given direction and then back to the starting one as if it was connected to the starting position via an elastic.
getter: A delegate that returns the value of the property to tween. Can be written as a lambda like this:
()=> myValue
where myValue is the name of the property to tween.
setter: A delegate that sets the value of the property to tween. Can be written as a lambda like this: x=> myValue = x
where myValue is the name of the property to tween.
direction: The direction and strength of the punch.
duration: The duration of the tween.
vibrato: Indicates how much the punch will vibrate.
elasticity: Represents how much (0 to 1) the vector will go beyond the starting position when bouncing backwards. 1 creates a full oscillation between the direction and the opposite decaying direction, while 0 oscillates only between the starting position and the decaying direction.

```
// Punch upwards a Vector3 called myVector in 1 second
DOTween.Punch(()=> myVector, x=> myVector = x, Vector3.up, 1);
```

static DOTween.Shake(getter, setter, float duration, float/Vector3 strength, int vibrato, float randomness, bool ignoreZAxis)
No FROM version.
Shakes a Vector3 along its X Y axes with the given values.
getter: A delegate that returns the value of the property to tween. Can be written as a lambda like this: ()=> myValue
where: myValue is the name of the property to tween.
setter: A delegate that sets the value of the property to tween. Can be written as a lambda like this:
x=> myValue = x
where myValue is the name of the property to tween.
duration: The duration of the tween.
strength: The shake strength. Using a Vector3 instead of a float lets you choose the strength

for each axis. vibrato Indicates how much will the shake vibrate.
randomness: Indicates how much the shake will be random (0 to 360 - values higher than 90 kind of suck, so beware). Setting it to 0 will shake along a single direction and behave like a random punch.
ignoreZAxis: If TRUE shakes only along the X Y axis (not available if you use a Vector3 for strength).
// Shake a Vector3 called myVector in 1 second
DOTween.Shake(()=> myVector, x=> myVector = x, 1, 5, 10, 45, false);

static DOTween.ToArray(getter, setter, float to, float duration)
No FROM version.
Tweens a Vector3 to the given end values. Ease is applied between each segment and not as a whole.
getter: A delegate that returns the value of the property to tween. Can be written as a lambda like this: ()=> myValue
where myValue is the name of the property to tween.
setter: A delegate that sets the value of the property to tween. Can be written as a lambda like this:
x=> myValue = x
where myValue is the name of the property to tween.
endValues: The end values to reach for each segment. This array must have the same length as durations.
durations: The duration of each segment. This array must have the same length as endValues.
Examples
// Tween a Vector3 between 3 values, for 1 second each
Vector3[] endValues = new[] { new Vector3(1,0,1), new Vector3(2,0,2), new Vector3(1,4,1) };
float[] durations = new[] { 1, 1, 1 };
DOTween.ToArray(()=> myVector, x=> myVector = x, endValues, durations);

static DOTween.ToAxis(getter, setter, float to, float duration, AxisConstraint axis)
Tweens a single axis of a Vector3 from its current value to the given one.
getter: A delegate that returns the value of the property to tween. Can be written as a lambda like this:
()=> myValue
where myValue is the name of the property to tween.
setter: A delegate that sets the value of the property to tween. Can be written as a lambda like this:
x=> myValue = x
where myValue is the name of the property to tween.
to: The end value to reach.
duration: The duration of the tween.

axis: The axis to tween.
// Tween the X of a Vector3 called myVector to 3 in 1 second
DOTween.ToAxis(()=> myVector, x=> myVector = x, 3, 1);
// Same as above, but tween the Y axis
DOTween.ToAxis(()=> myVector, x=> myVector = x, 3, 1, AxisConstraint.Y);

Virtual Tween

static DOTween.To(setter, float startValue, float endValue, float duration)
Tweens a virtual property from the given start to the given end value and implements a setter that allows to use that value with an external method or a lambda.
setter: The action to perform with the tweened value.
startValue: The value to start from.
endValue: The value to reach.
duration: The duration of the virtual tween.

DOTween.To(MyMethod, 0, 12, 0.5f);
// Where MyMethod is a function that accepts a float parameter
// (which will be the result of the virtual tween)

// Alternatively, with a lambda
DOTween.To(x => someProperty = x, 0, 12, 0.5f);

Creating a Sequence

Sequences are like Tweeners, but instead of animating a property or value they animate other Tweeners or Sequences as a group.
Sequences can be contained inside other Sequences without any limit to the depth of the hierarchy. The sequenced tweens don't have to be one after each other. You can overlap tweens with the Insert method. A tween (Sequence or Tweener) can be nested only inside a single other Sequence, meaning you can't reuse the same tween in multiple Sequences.
Also, the main Sequence will take control of all its nested elements, and you won't be able to control nested tweens separately (consider the Sequence as a movie timeline which becomes fixed once it starts up the for the very first time).
Finally, note that tweens added to a Sequence can't have infinite loops (but the root Sequence can).
IMPORTANT: don't use empty Sequences.
To create a Sequence, you follow these two steps:

1. Grab a new Sequence to use and store it as a reference

static DOTween.Sequence()
Returns a usable Sequence which you can store and add tweens to.

Sequence mySequence = DOTween.Sequence();

2. Add tweens, intervals and callbacks to your Sequence

Note that all these methods need to be applied before the Sequence starts (usually the next frame after you create it, unless it's paused), or they won't have any effect.
Also note that any nested Tweener/Sequence needs to be fully created before adding it to a Sequence, because after that it will be locked.
Delays and loops (when not infinite) will work even inside nested tweens.

Append(Tween tween)
Adds the given tween to the end of the Sequence (meaning the current total duration of the Sequence).
mySequence.Append(transform.DOMoveX(45, 1));

AppendCallback(TweenCallback callback)
Adds the given callback to the end of the Sequence (meaning the current total duration of the Sequence).
mySequence.AppendCallback(MyCallback);

AppendInterval(float interval)
Adds the given interval to the end of the Sequence (meaning the current total duration of the Sequence).
mySequence.AppendInterval(interval);

Insert(float atPosition, Tween tween)
Inserts the given tween at the given time position, thus allowing you to overlap tweens instead of just placing them one after each other.

mySequence.Insert(1, transform.DOMoveX(45, 1));
InsertCallback(float atPosition, TweenCallback callback)
Inserts the given callback at the given time position.

mySequence.InsertCallback(1, MyCallback);
Join(Tween tween)
Inserts the given tween at the same time position of the last tween or callback added to the Sequence.
// The rotation tween will be played together with the movement tween

mySequence.Append(transform.DOMoveX(45, 1));
mySequence.Join(transform.DORotate(new Vector3(0,180,0), 1));

Prepend(Tween tween)
Adds the given tween to the beginning of the Sequence, pushing forward in time the rest of

the contents.

mySequence.Prepend(transform.DOMoveX(45, 1));
PrependCallback(TweenCallback callback)
Adds the given callback to the beginning of the Sequence.

mySequence.PrependCallback(MyCallback);
PrependInterval(float interval)
Adds the given interval to the beginning of the Sequence, pushing forward in time the rest of the contents.

mySequence.PrependInterval(interval);
TIP: You can create Sequences made only of callbacks and use them as timers or stuff like that.

Examples

Creating a Sequence

```
// Grab a free Sequence to use
Sequence mySequence = DOTween.Sequence();
// Add a movement tween at the beginning
mySequence.Append(transform.DOMoveX(45, 1));
// Add a rotation tween as soon as the previous one is finished
mySequence.Append(transform.DORotate(new Vector3(0,180,0), 1));
// Delay the whole Sequence by 1 second
mySequence.PrependInterval(1);
// Insert a scale tween for the whole duration of the Sequence
mySequence.Insert(0, transform.DOScale(new Vector3(3,3,3), mySequence.Duration()));
```

Same as the previous example but with chaining (plus line breaks to make things clearer):

```
Sequence mySequence = DOTween.Sequence();
mySequence.Append(transform.DOMoveX(45, 1))
 .Append(transform.DORotate(new Vector3(0,180,0), 1))
 .PrependInterval(1)
 .Insert(0, transform.DOScale(new Vector3(3,3,3), mySequence.Duration()));
```

Settings, options and callbacks

DOTween uses a chaining approach when it comes to applying settings to a tween. Or you can change the global default options that will be applied to all newly created tweens.

Global settings

General settings

static LogBehaviour DOTween.logBehaviour
Default: LogBehaviour.ErrorsOnly
Depending on the chosen mode DOTween will log only errors, errors and warnings, or everything plus additional informations.
LogBehaviour.ErrorsOnly: Logs errors and nothing else.
LogBehaviour.Default: Logs errors and warnings.
LogBehaviour.Verbose: Logs errors, warnings and additional information.

static bool DOTween.maxSmoothUnscaledTime
Default: 0.15f
If useSmoothDeltaTime is TRUE, indicates the max time that will be considered as elapsed in case of timeIndependent tweens.

static bool DOTween.nestedTweenFailureBehaviour
Default: NestedTweenFailureBehaviour.TryToPreserveSequence
Behaviour in case safe mode is active and a tween nested inside a Sequence fails.

static bool DOTween.onWillLog
Used to intercept DOTween's logs. If this method isn't NULL, DOTween will call it before writing a log via Unity's own Debug log methods.
Return TRUE if you want DOTween to proceed with the log, FALSE otherwise.
This method must return a bool and accept two parameters: LogType (the type of Unity log that DOTween is trying to log), object (the message DOTween wants to log).

static bool DOTween.showUnityEditorReport
Default: FALSE
If set to TRUE you will get a DOTween report when exiting play mode (only in the Editor). Useful to know how many max Tweeners and Sequences you reached and optimize your final project accordingly.
Beware, this will slightly slow down your performance while inside Unity Editor.

static float DOTween.timeScale
Default: 1
Global timeScale applied to all tweens, both regular and independent.

static float DOTween.unscaledTimeScale
Default: 1
Global timeScale applied only to independent tweens.

static bool DOTween.useSafeMode
Default: TRUE
If set to TRUE tweens will be slightly slower but safer, allowing DOTween to automatically

take care of things like targets being destroyed while a tween is running.
Setting it to FALSE means you'll have to personally take care of killing a tween before its target is destroyed or somehow rendered invalid.
WARNING: on iOS safeMode works only if stripping level is set to "Strip Assemblies" or Script Call Optimization is set to "Slow and Safe", while on Windows 10 WSA it won't work if Master Configuration and .NET are selected.

static bool DOTween.useSmoothDeltaTime
Default: FALSE
If TRUE, DOTween will use Time.smoothDeltaTime instead of Time.deltaTime for UpdateType.Normal and UpdateType.Late tweens (unless they're set as timeScaleIndependent, in which case a value between the last timestep and maxSmoothUnscaledTime will be used instead). Setting this to TRUE will lead to smoother animations.

static DOTween.SetTweensCapacity(int maxTweeners, int maxSequences)
In order to be faster DOTween limits the max amount of active tweens you can have. If you go beyond that limit don't worry: it is automatically increased. Still, if you already know you'll need more (or less) than the default max Tweeners/Sequences (which is 200 Tweeners and 50 Sequences) you can set DOTween's capacity at startup and avoid hiccups when it's raised automatically. // Set max Tweeners to 2000 and max Sequences to 100 DOTween.SetTweensCapacity(2000, 100);

Settings applied to all newly created tweens

static bool DOTween.defaultAutoKill
Default: TRUE
Default autoKill behaviour applied to all newly created tweens.

static AutoPlay DOTween.defaultAutoPlay
Default: AutoPlay.All
Default autoPlay behaviour applied to all newly created tweens.

static float DOTween.defaultEaseOvershootOrAmplitude
Default: 1.70158f
Default overshoot/amplitude used for eases.

static float DOTween.defaultEasePeriod
Default: 0
Default period used for eases.

static Ease DOTween.defaultEaseType
Default: Ease.OutQuad
Default ease applied to all newly created Tweeners.

static LoopType DOTween.defaultLoopType
Default: LoopType.Restart
Default loop type applied to all newly created tweens that involve loops.

static bool DOTween.defaultRecyclable
Default: false
Default recycling behaviour applied to all newly created tweens.

static bool DOTween.defaultTimeScaleIndependent
Default: false
Sets whether Unity's timeScale/maximumDeltaTime should be taken into account by default or not.

static UpdateType DOTween.defaultUpdateType
Default: UpdateType.Normal
Default UpdateType applied to all newly created tweens.

Tweener and Sequence settings

Instance properties

float timeScale
Default: 1
Sets the internal timeScale for the tween.
// Sets the tween so that it plays at half the speed
myTween.timeScale = 0.5f;
TIP: This value can be tweened by another tween to achieve smooth slow-motion effects.

Chained settings

These settings can be chained to all types of tweens.
You can also chain them while a tween is running (except for SetAs, SetInverted, SetLoops and SetRelative)

SetAs(Tween tween \ TweenParams tweenParams)
Sets the parameters of the tween (id, ease, loops, delay, timeScale, callbacks, etc) as the parameters of the given one (doesn't copy specific SetOptions settings: those will need to be applied manually each time) or of the given TweenParams object.
Has no effect if the tween has already started.

transform.DOMoveX(4, 1).SetAs(myOtherTween);
SetAutoKill(bool autoKillOnCompletion = true)

If autoKillOnCompletion is set to TRUE the tween will be killed as soon as it completes,

otherwise it will stay in memory and you'll be able to reuse it.
NOTE: by default tweens are automatically killed at completion (so you need to use this method only if you plan to use FALSE as a parameter), but you can change the default behaviour in DOTween's Utility panel.

transform.DOMoveX(4, 1).SetAutoKill(false);
SetEase(Ease easeType \ AnimationCurve animCurve \ EaseFunction customEase)
Sets the ease of the tween.
If applied to a Sequence instead of a Tweener, the ease will be applied to the whole Sequence as if it was a single animated timeline. Sequences always have Ease.Linear by default, independently of the global default ease settings. You can pass it either a default ease (Ease – to see how default ease curves look, check out easings.net), an AnimationCurve or a custom ease function (see example).
Additionally, the following optional parameters can be set: they work only with Back and Elastic eases.
NOTE: Back and Elastic eases (meaning any ease that goes below or beyond the start and end values) don't work with paths.
overshoot: Eventual overshoot to use with Back ease (default is 1.70158), or number of flashes to use with Flash ease.
amplitude: Eventual amplitude to use with Elastic ease (default is 1.70158).
period: Eventual period to use with Elastic ease (default is 0), or power to use with Flash ease.
transform.DOMoveX(4, 1).SetEase(Ease.InOutQuint);
transform.DOMoveX(4, 1).SetEase(myAnimationCurve);
transform.DOMoveX(4, 1).SetEase(MyEaseFunction);

SPECIAL EASES
Flash, InFlash, OutFlash, InOutFlash: these eases will apply a flashing effect to the property you tween. The image below should be pretty clear about that.
overshoot Indicates the total number of flashes to apply. An even number will end the tween on the starting value, while an odd one will end it on the end value.
period Indicates the power in time of the ease, and must be between -1 and 1.
0 is balanced, 1 fully weakens the ease in time, -1 starts the ease fully weakened and gives it power towards the end.

EXTRA: EaseFactory
EaseFactory.StopMotion is an extra layer you can add to your easings, making them behave as if they were playing in stop-motion (practically, they will simulate the given FPS while tweening). It can be applied as a wrapper to any ease.
EaseFactory.StopMotion(int fps, Ease\AnimationCurve\EaseFunction ease)

transform.DOMoveX(4, 1).SetEase(EaseFactory.StopMotion(5, Ease.InOutQuint));
SetId(object id)

Sets an ID for the tween (which can then be used as a filter with DOTween's static methods). It can be an int, a string, an object or whatever.
NOTE: using an int or string ID makes filtered operations much faster (where int is also faster than string).
transform.DOMoveX(4, 1).SetId("supertween");
SetInverted()

EXPERIMENTAL: inverts this tween, so that it will play from the end to the beginning (playing it backwards will actually play it from the beginning to the end).
NOTE: if you just want to play a tween backwards you can simply use tween.PlayBackwards.
Has no effect if the tween has already started or is nested inside a Sequence.

tween.SetInverted();
SetLink(GameObject target, LinkBehaviour linkBehaviour = LinkBehaviour.KillOnDestroy)

Links this tween to a GameObject and assigns a behaviour depending on its active state.
This will also cause the tween to be automatically killed when the GameObject is destroyed.
NOTE: has no effect if the tween is added to a Sequence.
target The link target (unrelated to the target set via SetTarget).
linkBehaviour The behaviour to use (LinkBehaviour.KillOnDestroy is always evaluated—and set by default—even if you choose another one).

transform.DOMoveX(4, 1).SetLink(aGameObject,
LinkBehaviour.PauseOnDisableRestartOnEnable);
SetLoops(int loops, LoopType loopType = LoopType.Restart)
Sets the looping options (Restart, Yoyo, Incremental) for the tween.
Has no effect if the tween has already started. Also, infinite loops will not be applied if the tween is inside a Sequence.
Setting loops to -1 will make the tween loop infinitely.
LoopType.Restart: When a loop ends it will restart from the beginning.
LoopType.Yoyo: When a loop ends it will play backwards until it completes another loop, then forward again, then backwards again, and so on and on and on.
LoopType.Incremental: Each time a loop ends the difference between its endValue and its startValue will be added to the endValue, thus creating tweens that increase their values with each loop cycle. This loop type works only with Tweeners.

transform.DOMoveX(4, 1).SetLoops(3, LoopType.Yoyo);
SetRecyclable(bool recyclable)
Sets the recycling behaviour for the tween. If you don't set it then the default value (set either via DOTween.Init or DOTween.defaultRecyclable) will be used.
recyclable: If TRUE the tween will be recycled after being killed, otherwise it will be destroyed.

transform.DOMoveX(4, 1).SetRecyclable(true);

SetRelative(bool isRelative = true)
If isRelative is TRUE sets the tween as relative (the endValue will be calculated as startValue + endValue instead of being used directly). In case of Sequences, sets all the nested tweens as relative.

IMPORTANT: Has no effect on From tweens, since in that case you directly choose if the tween is relative or not when chaining the From setting.
Has no effect if the tween has already started.

transform.DOMoveX(4, 1).SetRelative();
SetTarget(object target)
Sets the target for the tween (which can then be used as a filter with DOTween's static methods). This is useful when using the generic tween method, while shortcuts automatically set this to the target of your shortcut.
NOTE: use it with caution. If you just want to set an ID for the tween use SetId instead.

DOTween.To(()=> myInstance.aFloat, (x)=> myInstance.aFloat = x, 2.5f, 1).SetTarget(myInstance);
SetUpdate(UpdateType updateType, bool isIndependentUpdate = false)
Sets the type of update (Normal, Late or Fixed) for the tween and eventually tells it to ignore Unity's timeScale and maximumDeltaTime.
Has no effect if the tween is nested inside a Sequence (only the Sequence's eventual SetUpdate will count). updateType The UpdateType to use:
UpdateType.Normal: Updates every frame during Update calls.
UpdateType.Late: Updates every frame during LateUpdate calls.
UpdateType.Fixed: Updates using FixedUpdate calls.
UpdateType.Manual: Updates via manual DOTween.ManualUpdate calls.
isIndependentUpdate If TRUE the tween will ignore Unity's Time.timeScale and maximumDeltaTime.

NOTE: independentUpdate works also with UpdateType.Fixed but is not recommended in that case (because at timeScale 0 FixedUpdate won't run).
transform.DOMoveX(4, 1).SetUpdate(UpdateType.Late, true);

Chained callbacks

OnComplete(TweenCallback callback)
Sets a callback that will be fired the moment the tween reaches completion, all loops included.

transform.DOMoveX(4, 1).OnComplete(MyCallback);
OnKill(TweenCallback callback)
Sets a callback that will be fired the moment the tween is killed.

transform.DOMoveX(4, 1).OnKill(MyCallback);
OnPlay(TweenCallback callback)
Sets a callback that will be fired when the tween is set in a playing state, after any eventual delay. Also called each time the tween resumes playing from a paused state.

transform.DOMoveX(4, 1).OnPlay(MyCallback);
OnPause(TweenCallback callback)
Sets a callback that will be fired when the tween state changes from playing to paused. If the tween has autoKill set to FALSE, this is called also when the tween reaches completion.

transform.DOMoveX(4, 1).OnPause(MyCallback);
OnRewind(TweenCallback callback)
Sets a callback that will be fired when the tween is rewinded, either by calling Rewind or by reaching the start position while playing backwards.

NOTE: Rewinding a tween that is already rewinded will not fire this callback.
transform.DOMoveX(4, 1).OnRewind(MyCallback);

OnStart(TweenCallback callback)
Sets a callback that will be fired once when the tween starts (meaning when the tween is set in a playing state the first time, after any eventual delay).

transform.DOMoveX(4, 1).OnStart(MyCallback);
OnStepComplete(TweenCallback callback)
Sets a callback that will be fired each time the tween completes a single loop cycle (meaning that, if you set your loops to 3, OnStepComplete will be called 3 times, contrary to OnComplete which will be called only once at the very end).

transform.DOMoveX(4, 1).OnStepComplete(MyCallback);
OnUpdate(TweenCallback callback)
Sets a callback that will be fired every time the tween updates.

transform.DOMoveX(4, 1).OnUpdate(MyCallback);
OnWaypointChange(TweenCallback callback)
Sets a callback that will be fired when a path tween's current waypoint changes.
This is a special callback which, contrary to the other ones, needs to accept a parameter of type int (which will be the newly changed waypoint index).

```
void Start() {
 transform.DOPath(waypoints, 1).OnWaypointChange(MyCallback);
}
void MyCallback(int waypointIndex) {
 Debug.Log("Waypoint index changed to " + waypointIndex);
}
```

By the way, callbacks attached to nested tweens will still work in the correct order. If you

want to use a callback with parameters, lambdas come to the rescue: // Callback without parameters transform.DOMoveX(4, 1).OnComplete(MyCallback); // Callback with parameters transform.DOMoveX(4, 1).OnComplete(()=>MyCallback(someParam, someOtherParam));

Tweener-specific settings and options

From(bool isRelative = false)
Changes the Tweener to a FROM tween (instead of a regular TO tween), immediately sending the target to its given value and then tweening to what was its previous value.
Must be chained before any other setting, except tween specific options.
isRelative If TRUE sets the tween as relative (the FROM value will be calculated as currentValue + endValue instead of being used directly). With FROM tweens you need to use this parameter instead of SetRelative.

```
// Regular TO tween
transform.DOMoveX(2, 1);

// FROM tween
transform.DOMoveX(2, 1).From();

// FROM tween but with relative FROM value
transform.DOMoveX(2, 1).From(true);
```

From(T fromValue, bool setImmediately = true, bool isRelative = false)
Allows to set the starting value of a tween directly, instead of relying on the target's value when the tween starts.
Must be chained before any other setting, except tween specific options.
fromValue The value from which the tween will start.
setImmediately If TRUE the target will be immediately set to the fromValue, otherwise it will wait for the tween to start.
isRelative If TRUE sets the tween as relative (the FROM value will be calculated as currentValue + fromValue instead of being used directly). With FROM tweens you need to use this parameter instead of SetRelative.
SetDelay(float delay)

Sets a delayed startup for the tween.
Has no effect if the tween has already started.
If chained to a Sequence doesn't add a real delay, but simply prepends an interval at the beginning of the Sequence (same as PrependInterval). Use the SetDelay overload below to change this behaviour.

```
transform.DOMoveX(4, 1).SetDelay(1);
```

SetDelay(float delay, bool asPrependedIntervalIfSequence)

Sets a delayed startup for the tween with options to choose how the delay is applied in case of Sequences.
Has no effect if the tween has already started.
asPrependedIntervalIfSequence Only used by Sequences: If FALSE sets the delay as a one-time occurrence (defaults to this for Tweeners), otherwise as a Sequence interval which will repeat at the beginning of every loop cycle.

transform.DOMoveX(4, 1).SetDelay(1);
SetSpeedBased(bool isSpeedBased = true)
If isSpeedBased is TRUE sets the tween as speed based (the duration will represent the number of units/degrees the tween moves x second). NOTE: if you want your speed to be constant, also set the ease to Ease.Linear.
Has no effect if the tween has already started or if it is a Sequence (or a nested tween inside a Sequence).
transform.DOMoveX(4, 1).SetSpeedBased();

SetOptions

Some Tweeners have specific special options that will be available to you depending on the type of thing you're tweening. It's all automatic: if a Tweener has specific options you'll see a specific SetOptions methods present for that Tweener, otherwise you won't. It's magic!
Note that these options are usually available only when creating tweens via the generic way, while shortcuts have the same options already included in their main creation method.
The important thing to remember is that, while all other settings can be chained together in any order, SetOptions must be chained immediately after the tween creation function, or it won't be available anymore.

Generic Tweens Specific Options (already included in the corresponding tween shortcuts)

Color tween ➨ SetOptions(bool alphaOnly)
Sets specific options for the tween of a Color.
alphaOnly If TRUE only the alpha of the color will be tweened.

DOTween.To(()=> myColor, x=> myColor = x, new Color(1,1,1,0), 1).SetOptions(true);
float tween ➨ SetOptions(bool snapping)
Sets specific options for the tween of a float.
snapping: If TRUE values will smoothly snap to integers.

DOTween.To(()=> myFloat, x=> myFloat = x, 45, 1).SetOptions(true);
Quaternion tween ➨ SetOptions(bool useShortest360Route)
Sets specific options for the tween of a Quaternion array.
useShortest360Route If TRUE (default) the rotation will take the shortest route and will not

rotate more than 360°. If FALSE the rotation will be fully accounted. Is always FALSE if the tween is set as relative.

DOTween.To(()=> myQuaternion, x=> myQuaternion = x, new Vector3(0,180,0), 1).SetOptions(true);
Rect tween ➨ SetOptions(bool snapping)
Sets specific options for the tween of a Rect.
snapping If TRUE values will smoothly snap to integers.

DOTween.To(()=> myRect, x=> myRect = x, new Rect(0,0,10,10), 1).SetOptions(true);
String tween ➨ SetOptions(bool richTextEnabled, ScrambleMode scrambleMode = ScrambleMode.None, string scrambleChars = null)
Sets specific options for the tween of a string.
richTextEnabled: If TRUE (default), rich text will be interpreted correctly while animated, otherwise all tags will be considered as normal text.
scramble: The type of scramble mode to use, if any.
If different than ScrambleMode.None the string will appear from a random animation of characters, otherwise it will compose itself regularly.
None: (default): no scrambling will be applied.

All/Uppercase/Lowercase/Numerals: type of characters to be used while scrambling.
Custom: will use the custom characters in scrambleChars.

scrambleChars: A string containing the characters to use for custom scrambling. Use as many characters as possible (minimum 10) because DOTween uses a fast scramble mode which gives better results with more characters.

DOTween.To(()=> myString, x=> myString = x, "hello world", 1).SetOptions(true, ScrambleMode.All);
Vector2/3/4 tween ➨ SetOptions(AxisConstraint constraint, bool snapping)
Sets specific options for the tween of a Vector2/3/4.
constraint: Tells the tween to animate only the given axis.
snapping: If TRUE values will smoothly snap to integers (great for pixel perfect movement).

DOTween.To(()=> myVector, x=> myVector = x, new Vector3(2,2,2), 1).SetOptions(AxisConstraint.Y, true);
Vector3Array tween ➨ SetOptions(bool snapping)
Sets specific options for the tween of a Vector3 array.
snapping: If TRUE values will smoothly snap to integers (great for pixel perfect movement).

DOTween.ToArray(()=> myVector, x=> myVector = x, myEndValues, myDurations).SetOptions(true);
DOPath Specific Options
Path tween ➨ SetOptions(bool closePath, AxisConstraint lockPosition =

AxisConstraint.None, AxisConstraint lockRotation = AxisConstraint.None)
Sets specific options for a path tween (created via the DOPath shortcut).

closePath: If TRUE the path will be automatically closed.
lockPosition: The eventual movement axis to lock. You can input multiple axis if you separate them like this:
AxisConstrain.X | AxisConstraint.Y.
lockRotation: The eventual rotation axis to lock. You can input multiple axis if you separate them like this: AxisConstrain.X | AxisConstraint.Y.
transform.DOPath(path, 4f).SetOptions(true, AxisConstraint.X);
Path tween ➨ SetLookAt(Vector3 lookAtPosition/lookAtTarget/lookAhead, Vector3 forwardDirection, Vector3 up, bool stableZRotation)
Additional LookAt options for Path tweens (created via the DOPath shortcut). Depending on the overload you choose, it either: a) orients the target towards the given position, b) orients the target towards the given transform, c) orients the target to the path with the given lookAhead.
Must be chained directly to the tween creation method or to a SetOptions method.
lookAtPosition: The position to look at.
lookAtTarget: The target to look at.
lookAhead: The lookAhead percentage to use when orienting to the path (0 to 1).
forwardDirection: Optional overload: the eventual direction to consider as "forward".
Default: the regular forward side of the transform.
up Optiona overload: the vector that defines in which direction up is.
Default: Vector3.up
stableZRotation Optional overload: if set to TRUE doesn't rotate the target along the Z axis.
transform.DOPath(path, 4f).SetLookAt(new Vector3(2,1,3));
transform.DOPath(path, 4f).SetLookAt(someOtherTransform);
transform.DOPath(path, 4f).SetLookAt(0.01f);

DOPath Specific Options

If you used HOTween previously, you will know TweenParms (now called TweenParams): they're used to store settings that you can then apply to multiple tweens. The difference from HOTween is that they're not necessary at all, since now settings chaining is done directly on a tween. Instead they're here only as an extra utility class. To use it, create a new TweenParams instance or Clear() an existing one, then add settings like you do with regular tween chaining. To apply them to a tween you then use SetAs.
// Store settings for an infinite looping tween with elastic ease
TweenParams tParms = new TweenParams().SetLoops(-1).SetEase(Ease.OutElastic);
// Apply them to a couple of tweens
transformA.DOMoveX(15, 1).SetAs(tParms);
transformB.DOMoveY(10, 1).SetAs(tParms);

TweenParams

If you used HOTween previously, you will know TweenParms (now called TweenParams): they're used to store settings that you can then apply to multiple tweens. The difference from HOTween is that they're not necessary at all, since now settings chaining is done directly on a tween. Instead they're here only as an extra utility class. To use it, create a new TweenParams instance or Clear() an existing one, then add settings like you do with regular tween chaining. To apply them to a tween you then use SetAs.

```
// Store settings for an infinite looping tween with elastic ease
TweenParams tParms = new TweenParams().SetLoops(-1).SetEase(Ease.OutElastic);
// Apply them to a couple of tweens
transformA.DOMoveX(15, 1).SetAs(tParms);
transformB.DOMoveY(10, 1).SetAs(tParms);
```

More on chaining

Just to be clear, you don't need to chain one thing at a time and you can do this instead:

```
transform.DOMoveX(45, 1).SetDelay(2).SetEase(Ease.OutQuad).OnComplete(MyCallback);
```

Controlling a tween

You have 3 ways to manipulate a tween. All of them share the same method names, except for shortcut-enhanced ones which have an additional DO prefix.

A. Via static methods and filters
The DOTween class contains many static methods that allow you to control tweens.
Each of them comes both in an "All" version (like DOTween.KillAll), which applies to all existing tweens, and a simple version (DOTween.Kill(myTargetOrId)) with a parameter that allows you to filter operations by a tween's id or target (id is set manually via SetId, while target is set automatically when creating a tween via a shortcut). Static methods additionally return an int, which represents all the tweens that were actually able to perform the requested operation.

```
// Pauses all tweens
DOTween.PauseAll();
// Pauses all tweens that have "badoom" as an id
DOTween.Pause("badoom");
// Pauses all tweens that have someTransform as a target
DOTween.Pause(someTransform);
```

B. Directly from the tween
instead of using static methods you can just call the same methods from your tween's

reference. myTween.Pause();

C. From a shortcut-enhanced reference
Same as above, but you can call those same methods from a shortcut-enhanced object. Remember that in this case the method names have an additional DO prefix to distinguish them from the regular object methods. transform.DOPause();

Control methods

Again, remember that all these method names are shared by all manipulation ways, but that in case of object shortcuts there's an additional DO prefix.

IMPORTANT: remember that to use these methods on a tween after it has ended, you have to disable its autoKill behaviour, otherwise a tween is automatically killed at completion.

CompleteAll/Complete(bool withCallbacks = false) Sends a tween to its end position (has no effect with tweens that have infinite loops). withCallbacks For Sequences only: if TRUE internal Sequence callbacks will be fired, otherwise they will be ignored.

FlipAll/Flip()
Flips the direction of a tween (backwards if it was going forward or viceversa).
GotoAll/Goto(float to, bool andPlay = false)
Send the tween to the given position in time.
to Time position to reach (if higher than the whole tween duration the tween will simply reach its end). andPlay If TRUE the tween will play after reaching the given position, otherwise it will be paused.

KillAll/Kill(bool complete = false, params object[] idsOrTargetsToExclude)
Kills the tween.
A tween is killed automatically when it reaches completion (unless you prevent it using SetAutoKill(false)), but you can use this method to kill it sooner if you don't need it anymore. complete If TRUE instantly completes the tween (setting the target to its final value) before killing it. idsOrTargetsToExclude KillAll only > Eventual targets or ids to exclude from the operation.

PauseAll/Pause()
Pauses the tween.

PlayAll/Play()
Plays the tween.

PlayBackwardsAll/PlayBackwards()
Plays the tween backwards.

PlayForwardAll/PlayForward()

Plays the tween forward.

RestartAll/Restart(bool includeDelay = true, float changeDelayTo = -1)
Restarts the tween.
includeDelay: If TRUE includes the eventual tween delay, otherwise skips it.
changeDelayTo: Sets the tween's delay to the given one.
RewindAll/Rewind(bool includeDelay = true)
Rewinds: and pauses the tween.
includeDelay: If TRUE includes the eventual tween delay, otherwise skips it.
SmoothRewindAll/SmoothRewind()
Smoothly rewinds the tween (delays excluded).
A "smooth rewind" animates the tween to its start position (instead of jumping to it), skipping all elapsed loops (except in case of LoopType.Incremental) while keeping the animation fluent.
If called on a tween who is still waiting for its delay to happen, it will simply set the delay to 0 and pause the tween.
NOTE: A tween that was smoothly rewinded will have its play direction flipped.

TogglePauseAll/TogglePause()

Plays the tween if it was paused, pauses it if it was playing.

Special control methods

Common for all tweens

ForceInit()
Type-specific
These are special control methods that work only on some specific type of tweens

GotoWaypoint(int waypointIndex, bool andPlay = false)

Getting data from tweens

Static methods (DOTween)

static List PausedTweens()
Returns a list of all active tweens in a paused state, or NULL if there are no active paused tweens.
Beware: calling this method creates garbage allocation since a new List is generated with each call.

static List PlayingTweens()

Returns a list of all active tweens in a playing state, or NULL if there are no active playing tweens.
Beware: calling this method creates garbage allocation since a new List is generated with each call.

static List TweensById(object id, bool playingOnly = false)
Returns a list of all active tweens with the given id, or NULL if there are no active tweens with the given id.
Beware: calling this method creates garbage allocation since a new List is generated with each call.
playingOnly: If TRUE returns only the tweens with the given ID that are currently playing, otherwise all of them.

static List TweensByTarget(object target, bool playingOnly = false)
Returns a list of all active tweens with the given target, or NULL if there are no active tweens with the given target.

NOTE: a tween's target is set automatically when using shortcuts, not when using the generic way.

NOTE: the DOTweenAnimation Visual Editor will assign its gameObject as a target (instead than a transform, material, or whatever was the actual target of the shortcut), so use that if you want to grab visually created tweens.

Beware: calling this method creates garbage allocation since a new List is generated with each call. playingOnly If TRUE returns only the tweens with the given target that are currently playing, otherwise all of them.

static bool IsTweening(object idOrTarget, bool alsoCheckIfPlaying = false)
Returns TRUE if a tween with the given ID or target is active.

You can also use this to know if a shortcut tween is active on a target.

alsoCheckIfPlaying If FALSE (default) returns TRUE as long as a tween for the given target/ID is active, otherwise also requires it to be playing.

transform.DOMoveX(45, 1); // transform is automatically added as the tween target
DOTween.IsTweening(transform); // Returns TRUE

static int TotalActiveSequences()
Returns the total number of active Sequences, regardless if they're playing or not.

int totalActiveSequences = DOTween.TotalActiveSequences();

static int TotalActiveTweens()

Returns the total number of active tweens (including both Sequences and Tweeners), regardless if they're playing or not.

int totalActive = DOTween.TotalActiveTweens();

static int TotalActiveTweeners()
Returns the total number of active Tweeners, regardless if they're playing or not.

int totalActiveTweeners = DOTween.TotalActiveTweeners();
static int TotalPlayingTweens()
Returns the total number of active and playing tweens. A tween is considered as playing even if its delay is actually playing.

int totalPlaying = DOTween.TotalPlayingTweens();
static int TotalTweensById()
Returns the total number of active tweens with the given ID, regardless if they're playing or not.
int totalTweensWithId = DOTween.TotalTweensById();

Instance methods (Tween/Tweener/Sequence)

float fullPosition
Gets and sets the time position (loops included, delays excluded) of the tween.

int CompletedLoops()
Returns the total number of loops completed by the tween.

int completedLoops = myTween.CompletedLoops();
float Delay()
Returns the eventual delay set for the tween.

float eventualDelay = myTween.Delay();
float Duration(bool includeLoops = true)
Returns the duration of the tween (delays excluded, loops included if includeLoops is TRUE).
NOTE: when using settings like SpeedBased, the duration will be recalculated when the tween starts.
includeLoops: If TRUE returns the full duration loops included, otherwise the duration of a single loop cycle.

float loopCycleDuration = myTween.Duration(false);
float fullDuration = myTween.Duration();
float Elapsed(bool includeLoops = true)
Returns the currently elapsed time for the tween (delays excluded, loops included if includeLoops is TRUE).

includeLoops If TRUE returns the full elapsed time since startup loops included, otherwise the elapsed time within the current loop cycle.

float loopCycleElapsed = myTween.Elapsed(false);
float fullElapsed = myTween.Elapsed();
float ElapsedDirectionalPercentage()
Returns the elapsed percentage (0 to 1) of this tween (delays excluded), based on a single loop, and calculating eventual backwards Yoyo loops as 1 to 0 instead of 0 to 1.
float ElapsedPercentage(bool includeLoops = true)
Returns the elapsed percentage (0 to 1) of this tween (delays excluded, loops included if includeLoops is TRUE).
includeLoops: If TRUE returns the elapsed percentage since startup loops included, otherwise the elapsed percentage within the current loop cycle.

float loopCycleElapsedPerc = myTween.ElapsedPercentage(false);
float fullElapsedPerc = myTween.ElapsedPercentage();
bool IsActive()
Returns FALSE if the tween has been killed.

bool isActive = myTween.IsActive();
bool IsBackwards()
Returns TRUE if the tween was reversed and is set to go backwards.

bool isBackwards = myTween.IsBackwards(); bool IsComplete()
Returns TRUE if the tween is complete (silently fails and returns FALSE if the tween has been killed).

bool isComplete = myTween.IsComplete();
bool IsInitialized()
Returns TRUE if this tween has been initialized.

bool isInitialized = myTween.IsInitialized();
bool IsPlaying()
Returns TRUE if the tween is playing.

bool isPlaying = myTween.IsPlaying();
bool IsTimeScaleIndependent()
Returns TRUE if the tween was set to be timeScale independent via the tween.SetUpdate method.

bool isTimeScaleIndependent = myTween.IsTimeScaleIndependent();
int Loops()
Returns the total number of loops assigned to the tween.
int totLoops = myTween.Loops();

Instance methods ➡ Path tweens

Vector3 PathGetPoint(float pathPercentage)
Returns a point on a path based on the given path percentage (returns Vector3.zero if this is not a path tween, if the tween is invalid, or if the path is not yet initialized).
A path is initialized after its tween starts, or immediately if the tween was created with the Path Editor (DOTween Pro feature).
You can force a path to be initialized by calling ForceInit.
pathPercentage Percentage of the path (0 to 1) on which to get the point.

Vector3 myPathMidPoint = myTween.PathGetPoint(0.5f);
Vector3[] PathGetDrawPoints(int subdivisionsXSegment = 10)
Returns an array of points that can be used to draw the path (returns NULL if this is not a path tween, if the tween is invalid, or if the path is not yet initialized).
NOTE: This method generates allocations, because it creates a new array.
A path is initialized after its tween starts, or immediately if the tween was created with the Path Editor (DOTween Pro feature).
You can force a path to be initialized by calling ForceInit.
subdivisionsXSegment How many points to create for each path segment (waypoint to waypoint). Only used in case of non-Linear paths

Vector3[] myPathDrawPoints = myTween.PathGetDrawPoints(); float PathLength()
Returns the length of a path (returns -1 if this is not a path tween, if the tween is invalid, or if the path is not yet initialized).
A path is initialized after its tween starts, or immediately if the tween was created with the Path Editor (DOTween Pro feature).
You can force a path to be initialized by calling ForceInit.
float myPathLength = myTween.PathLength();

WaitFor coroutines/Tasks

Coroutines

Tweens come with a useful set of YieldInstructions which you can place inside your Coroutines, and that allow you to wait for something to happen. All these methods have an optional bool parameter that allows to return a CustomYieldInstruction. WaitForCompletion() Creates a yield instruction that waits until the tween is killed or complete. IEnumerator SomeCoroutine() { Tween myTween = transform.DOMoveX(45, 1); yield return myTween.WaitForCompletion(); // This log will happen after the tween has completed Debug.Log("Tween completed!"); } WaitForElapsedLoops(int elapsedLoops) Creates a yield instruction that waits until the tween is killed or has gone through the given amount of loops. IEnumerator SomeCoroutine() { Tween myTween = transform.DOMoveX(45,

1).SetLoops(4); yield return myTween.WaitForElapsedLoops(2); // This log will happen after the 2nd loop has finished Debug.Log("Tween has looped twice!"); } WaitForKill() Creates a yield instruction that waits until the tween is killed. IEnumerator SomeCoroutine() { Tween myTween = transform.DOMoveX(45, 1); yield return myTween.WaitForKill(); // This log will happen after the tween has been killed Debug.Log("Tween killed!"); } WaitForPosition(float position) Creates a yield instruction that waits until the tween is killed or has reached the given time position (loops included, delays excluded). IEnumerator SomeCoroutine() { Tween myTween = transform.DOMoveX(45, 1); yield return myTween.WaitForPosition(0.3f); // This log will happen after the tween has played for 0.3 seconds Debug.Log("Tween has played for 0.3 seconds!"); } WaitForRewind() Creates a yield instruction that waits until the tween is killed or rewinded. IEnumerator SomeCoroutine() { Tween myTween = transform.DOMoveX(45, 1).SetAutoKill(false).OnComplete(myTween.Rewind); yield return myTween.WaitForRewind(); // This log will happen when the tween has been rewinded Debug.Log("Tween rewinded!"); } WaitForStart() Creates a yield instruction that waits until the tween is killed or started (meaning when the tween is set in a playing state the first time, after any eventual delay). IEnumerator SomeCoroutine() { Tween myTween = transform.DOMoveX(45, 1); yield return myTween.WaitForStart(); // This log will happen when the tween starts Debug.Log("Tween started!"); }

Tasks

Requires at least Unity 2018.1 and .NET Standard 2.0 or 4.6
These methods return a Task to be used inside async operations, to wait for something to happen.

AsyncWaitForCompletion()
Returns a Task that waits until the tween is killed or complete.
await myTween.AsyncWaitForCompletion();

AsyncWaitForElapsedLoops(int elapsedLoops)
Returns a Task that waits until the tween is killed or has gone through the given amount of loops.
await myTween.AsyncWaitForElapsedLoops();

AsyncWaitForKill()
Returns a Task that waits until the tween is killed.
await myTween.AsyncWaitForKill();

AsyncWaitForPosition(float position)
Returns a Task that waits until the tween is killed or has reached the given time position (loops included, delays excluded).
await myTween.AsyncWaitForPosition(0.3f);

AsyncWaitForRewind()
Returns a Task that waits until the tween is killed or rewinded.
await myTween.AsyncWaitForRewind();

AsyncWaitForStart()
Returns a Task that waits until the tween is killed or started (meaning when the tween is set in a playing state the first time, after any eventual delay).
await myTween.AsyncWaitForStart();

Additional methods

Static methods (DOTween)

static DOTween.Clear(bool destroy = false)
Kills all tweens, clears all pools, resets the max Tweeners/Sequences capacities to the default values.
NOTE: this method should be used only for debug purposes or when you don't plan to create/run any more tweens in your project, because it completely deinitializes DOTween and its internal plugins. If you simply want to kill all tweens use the static method DOTween.KillAll() instead.
destroy: If TRUE also destroys DOTween's gameObject and resets its initializiation, default settings and everything else (so that next time you use it it will need to be re-initialized).
static DOTween.ClearCachedTweens()
Clears all cached tween pools.
static DOTween.Validate()
Validates all active tweens and removes eventually invalid ones (usually because their target was destroyed). This is a slightly expensive operation so use it with care. Also, no need to use it at all especially if safe mode is ON.
static DOTween.ManualUpdate(float deltaTime, float unscaledDeltaTime)
Updates all tweens that are set to UpdateType.Manual.

Instance methods (Tween/Tweener/Sequence)

Tween (Tweeners + Sequences)

Tween Done()
Optional: indicates that the tween creation has ended. To be used (optionally) as the last element of tween chaining creation.
This method won't do anything except in case of 0-duration tweens, where it will complete them immediately instead of waiting for the next internal update routine (unless they're nested in a Sequence, in which case the Sequence will still be the one in control and this method will be ignored).

ManualUpdate(float deltaTime, float unscaledDeltaTime)

Forces this tween to update manually, regardless of the UpdateType set via SetUpdate.

NOTE: the tween will still be subject to normal tween rules, so if for example it's paused this method will do nothing. Also note that if you only want to update this tween instance manually you'll have to set it to UpdateType.Manual anyway, so that it's not updated automatically.
deltaTime Manual deltaTime.
unscaledDeltaTime Unscaled delta time (used with tweens set as timeScaleIndependent).

Tweener

ChangeEndValue(newEndValue, float duration = -1, bool snapStartValue = false)
Changes the end value of a Tweener and rewinds it (without pausing it).
Has no effect with Tweeners that are inside Sequences.
NOTE: works on regular tweens, not on ones that do more than just animate one value from one point to another (like DOLookAt).
NOTE: for shortcuts that accept a single axis (DOMoveX/Y/Z, DOScaleX/YZ, etc) you still have to pass a full Vector2/3/4 value, even if only the one you set in the tween will be considered. newEndValue New end value.
duration: If bigger than 0 also changes the duration of the tween.
snapStartValue If TRUE the start value will become the current target's value, otherwise it will stay the same.
ChangeStartValue(newStartValue, float duration = -1)
Changes the start value of a Tweener and rewinds it (without pausing it).
Has no effect with Tweeners that are inside Sequences.
NOTE: works on regular tweens, not on ones that do more than just animate one value from one point to another (like DOLookAt).
NOTE: for shortcuts that accept a single axis (DOMoveX/Y/Z, DOScaleX/YZ, etc) you still have to pass a full Vector2/3/4 value, even if only the one you set in the tween will be considered. newStartValue New start value.
duration: If bigger than 0 also changes the duration of the tween.

ChangeValues(newStartValue, newEndValue, float duration = -1)
Changes the start and end values of a Tweener and rewinds it (without pausing it).
Has no effect with Tweeners that are inside Sequences.
NOTE: works on regular tweens, not on ones that do more than just animate one value from one point to another (like DOLookAt).
NOTE: for shortcuts that accept a single axis (DOMoveX/Y/Z, DOScaleX/YZ, etc) you still have to pass a full Vector2/3/4 value, even if only the one you set in the tween will be considered.
newStartValue New start value.

newEndValue New end value.
duration If bigger than 0 also changes the duration of the tween.

Editor methods

Previewing tweens in editor

static DOTweenEditorPreview.PrepareTweenForPreview(bool clearCallbacks = true, bool preventAutoKill = true, bool andPlay = true)

Readies the given tween for editor preview by setting its UpdateType to Manual plus eventual extra settings.
clearCallbacks If TRUE (recommended) removes all callbacks (OnComplete/Rewind/etc) when previewing.
preventAutoKill If TRUE prevents the tween from being auto-killed at completion when previewing.
andPlay If TRUE starts playing the tween immediately.

static DOTweenEditorPreview.Start(Action onPreviewUpdated = null)
Starts the update loop of tweens in the editor. Has no effect during playMode.
Before calling this method you must add tweens to the preview loop via DOTweenEditorPreview.PrepareTweenForPreview.
onPreviewUpdated Eventual callback to call after every update (while in editor preview).
static DOTweenEditorPreview.Stop()
Stops the preview update loop and clears any callback.

Virtual + extra methods

Virtual methods

static Tweener DOVirtual.Float(float from, float to, float duration, TweenCallback onVirtualUpdate) Tweens a virtual float.

You can add regular settings to the generated tween, but do not use OnUpdate or you will overwrite the onVirtualUpdate parameter.

from: The value to start from.
to: The value to tween to. duration: The duration of the tween. onVirtualUpdate: A callback which must accept a parameter of type float, called at each update.
static Tweener DOVirtual.Int(int from, int to, float duration, TweenCallback onVirtualUpdate) Tweens a virtual integer.
You can add regular settings to the generated tween, but do not use OnUpdate or you will

overwrite the onVirtualUpdate parameter.
from: The value to start from.
to: The value to tween to.
duration: The duration of the tween.
onVirtualUpdate: A callback which must accept a parameter of type int, called at each update.
static Tweener DOVirtual.Vector3(Vector3 from, Vector3 to, float duration, TweenCallback onVirtualUpdate) Tweens a virtual Vector3.
You can add regular settings to the generated tween, but do not use OnUpdate or you will overwrite the onVirtualUpdate parameter.
from: The value to start from.
to: The value to tween to.
duration: The duration of the tween.
onVirtualUpdate: A callback which must accept a parameter of type Vector3, called at each update.
static Tweener DOVirtual.Color(Color from, Color to, float duration, TweenCallback onVirtualUpdate) Tweens a virtual Color.
You can add regular settings to the generated tween, but do not use OnUpdate or you will overwrite the onVirtualUpdate parameter.
from: The value to start from.
to: The value to tween to.
duration: The duration of the tween.
onVirtualUpdate: A callback which must accept a parameter of type Color, called at each update.
static float DOVirtual.EasedValue(float from, float to, float lifetimePercentage, Ease easeType \ AnimationCurve animCurve)
Returns a value based on the given ease and lifetime percentage (0 to 1).

Comes with various overloads to allow an AnimationCurve ease or custom overshoot/period/amplitude.

from: The value to start from when lifetimePercentage is 0.
to: The value to reach when lifetimePercentage is 1.
lifetimePercentage: The time percentage (0 to 1) at which the value should be taken.
easeType: The type of ease.
static float DOVirtual.EasedValue(Vector3 from, Vector3 to, float lifetimePercentage, Ease easeType \ AnimationCurve animCurve)
Returns a value based on the given ease and lifetime percentage (0 to 1).
Comes with various overloads to allow an AnimationCurve ease or custom overshoot/period/amplitude. from: The value to start from when lifetimePercentage is 0.
to: The value to reach when lifetimePercentage is 1.
lifetimePercentage: The time percentage (0 to 1) at which the value should be taken.

easeType The type of ease.
static Tween DOVirtual.DelayedCall(float delay, TweenCallback callback, bool ignoreTimeScale = true) Fires the given callback after the given time.
Returns a Tween so you can eventually store it and pause/kill/etc it.
delay Callback delay.
callback Callback to fire when the delay has expired.
ignoreTimeScale If TRUE (default) ignores Unity's timeScale.
// Example 1: calling another method after 1 second
DOVirtual.DelayedCall(1, MyOtherMethodName);
// Example 2: using a lambda to throw a log after 1 second
DOVirtual.DelayedCall(1, ()=> Debug.Log("Hello world"));

Extra methods

static Vector3 DOCurve.CubicBezier.GetPointOnSegment(Vector3 startPoint, Vector3 startControlPoint, Vector3 endPoint, Vector3 endControlPoint, float factor)
Calculates a point along the given Cubic Bezier segment-curve.
startPoint Segment start point.
startControlPoint Start point's control point/handle.
endPoint Segment end point.
endControlPoint End point's control point/handle.
factor 0-1 percentage along which to retrieve point.

static Vector3 DOCurve.CubicBezier.GetSegmentPointCloud(Vector3 startPoint, Vector3 startControlPoint, Vector3 endPoint, Vector3 endControlPoint, int resolution = 10)
Returns an array containing a series of points along the given Cubic Bezier segment-curve.
startPoint Segment start point.
startControlPoint Start point's control point/handle.
endPoint Segment end point.
endControlPoint End point's control point/handle.
resolution Cloud resolution (min: 2).

static Vector3 DOCurve.CubicBezier.GetSegmentPointCloud(List addToList, Vector3 startPoint, Vector3 startControlPoint, Vector3 endPoint, Vector3 endControlPoint, int resolution = 10)
Calculates a series of points along the given Cubic Bezier segment-curve and adds them to the given list.
addToList Existing list to which the points should be added (note that they will be literally added, without clearing the list first).
startPoint Segment start point.
startControlPoint Start point's control point/handle.
endPoint Segment end point.

endControlPoint End point's control point/handle.
resolution Cloud resolution (min: 2).

Creating custom plugins

The sample UnityPackage from the examples page shows, among other things, how to create custom plugins.