```swift
import DeveloperToolsSupport
import Foundation
import PencilKit.PKContentVersion
import PencilKit.PKDrawing
import PencilKit.PKEraserTool
import PencilKit.PKFloatRange
import PencilKit.PKInk
import PencilKit.PKInkType
import PencilKit.PKInkingTool
import PencilKit.PKLassoTool
import PencilKit.PKStroke
import PencilKit.PKStrokePath
import PencilKit.PKStrokePoint
import PencilKit.PKTool
import _Concurrency
import _StringProcessing
import _SwiftConcurrencyShims

@available(iOS 13.0, macOS 10.15, *)
public struct PKDrawing {

    public init()

    public init(data: Data) throws

    public func dataRepresentation() ->
Data

    /// The bounds of the drawing's
contents.
    /// If this bounds is used to render
an image, no contents will be cropped.
    public var bounds: CGRect { get }
    public func image(from rect: CGRect,
scale: CGFloat) -> NSImage

    /// Applies `transform` to the
contents of this drawing.
    ///
    /// - parameter transform: The
transform to apply to this drawing.
    public mutating func transform(using
transform: CGAffineTransform)

    /// Returns a new drawing with
`transform` applied.
    ///
    /// - parameter transform: The
transform to apply to this drawing.
    /// - returns: A new copy of this
drawing with `transform` applied.
    public func transformed(using
transform: CGAffineTransform) ->
PKDrawing

    /// Appends the contents of `drawing`
on top of the receiver's contents.
    ///
    /// - parameter drawing: The drawing
to append.
    public mutating func append(_
toAppend: PKDrawing)

    /// Returns a new drawing by
appending the contents of `drawing` on
```

```
the receiver’s contents.
    ///
    /// - parameter drawing: The drawing
to append.
    /// - returns: A new copy of this
drawing with `drawing` appended.
    public func appending(_ toAppend:
PKDrawing) -> PKDrawing

    /// The PencilKit version required to
use this drawing.
    @available(iOS 17.0, macOS 14.0, *)
    public var requiredContentVersion:
PKContentVersion { get }
}

@available(iOS 14.0, macOS 11.0, *)
extension PKDrawing {

    /// Create a new drawing with
`strokes`.
    public init<S>(strokes: S) where S :
Sequence, S.Element == PKStroke

    /// The strokes that this drawing
contains.
    public var strokes: [PKStroke]
}

@available(iOS 13.0, macOS 10.15, *)
extension PKDrawing : Equatable {

    /// Returns a Boolean value
```

```
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (a: PKDrawing,
b: PKDrawing) -> Bool
}

@available(iOS 13.0, macOS 10.15, *)
extension PKDrawing : Codable {

    /// Creates a new instance by
decoding from the given decoder.
    ///
    /// This initializer throws an error
if reading from the decoder fails, or
    /// if the data read is corrupted or
otherwise invalid.
    ///
    /// - Parameter decoder: The decoder
to read data from.
    public init(from decoder: any
Decoder) throws

    /// Encodes this value into the given
encoder.
```

```
    ///
    /// If the value fails to encode
anything, `encoder` will encode an empty
    /// keyed container in its place.
    ///
    /// This function throws an error if
any values are invalid for the given
    /// encoder's format.
    ///
    /// - Parameter encoder: The encoder
to write data to.
    public func encode(to encoder: any
Encoder) throws
}

@available(iOS 13.0, macOS 11.0, *)
public struct PKEraserTool : PKTool,
Equatable {

    /// Create a new eraser.
    ///
    /// - Parameter eraserType: The type
of eraser.
    public init(_ eraserType:
PKEraserTool.EraserType)

    /// Create a new fixed width bitmap
eraser tool.
    ///
    /// - Parameters:
    ///   - eraserType: The type of
eraser.
    ///   - width: The width of the
```

```
eraser.
    @available(iOS 16.4, macOS 13.3, *)
    public init(_ eraserType:
PKEraserTool.EraserType, width: CGFloat)

    public enum EraserType {

        case vector

        case bitmap

        @available(iOS 16.4, macOS 13.3,
*)
        case fixedWidthBitmap

        /// The default width for an
eraser type.
        @available(iOS 16.4, macOS 13.3,
*)
        public var defaultWidth: CGFloat
{ get }

        /// The valid width range for an
eraser type.
        @available(iOS 16.4, macOS 13.3,
*)
        public var validWidthRange:
ClosedRange<CGFloat> { get }

        /// Returns a Boolean value
indicating whether two values are equal.
        ///
        /// Equality is the inverse of
```

```
inequality. For any values `a` and `b`,
        /// `a == b` implies that `a !=
b` is `false`.
        ///
        /// - Parameters:
        ///   - lhs: A value to compare.
        ///   - rhs: Another value to
compare.
        public static func == (a:
PKEraserTool.EraserType, b:
PKEraserTool.EraserType) -> Bool

        /// Hashes the essential
components of this value by feeding them
into the
        /// given hasher.
        ///
        /// Implement this method to
conform to the `Hashable` protocol. The
        /// components used for hashing
must be the same as the components
compared
        /// in your type's `==` operator
implementation. Call `hasher.combine(_:)`
        /// with each of these
components.
        ///
        /// - Important: In your
implementation of `hash(into:)`,
        ///   don't call `finalize()` on
the `hasher` instance provided,
        ///   or replace it with a
different instance.
```

```
        ///   Doing so may become a
compile-time error in the future.
        ///
        /// - Parameter hasher: The
hasher to use when combining the
components
        ///   of this instance.
        public func hash(into hasher:
inout Hasher)

        /// The hash value.
        ///
        /// Hash values are not
guaranteed to be equal across different
executions of
        /// your program. Do not save
hash values to use during a future
execution.
        ///
        /// - Important: `hashValue` is
deprecated as a `Hashable` requirement.
To
        ///   conform to `Hashable`,
implement the `hash(into:)` requirement
instead.
        ///   The compiler provides an
implementation for `hashValue` for you.
        public var hashValue: Int { get }
    }

    /// The eraser type.
    public var eraserType:
PKEraserTool.EraserType
```

```
    /// The width of the eraser.
    @available(iOS 16.4, macOS 13.3, *)
    public var width: CGFloat

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (a:
PKEraserTool, b: PKEraserTool) -> Bool
}

@available(iOS 13.0, macOS 11.0, *)
extension PKEraserTool.EraserType :
Equatable {
}

@available(iOS 13.0, macOS 11.0, *)
extension PKEraserTool.EraserType :
Hashable {
}

/// PKInk provides a description of how
marks on a PKCanvas render and are
```

```
created.
@available(iOS 14.0, macOS 11.0, *)
public struct PKInk {

    public typealias InkType =
PKInkingTool.InkType

    public init(_ inkType: PKInk.InkType,
color: NSColor = NSColor.black)

    /// The type of ink.
    public var inkType: PKInk.InkType

    public var color: NSColor

    /// The PencilKit version required to
use this ink.
    @available(iOS 17.0, macOS 14.0, *)
    public var requiredContentVersion:
PKContentVersion { get }
}

@available(iOS 13.0, macOS 11.0, *)
public struct PKInkingTool : PKTool,
Equatable {

    public init(_ inkType:
PKInkingTool.InkType, color: NSColor =
NSColor.black, width: CGFloat? = nil)

    /// The type of ink mark that will be
made.
    public enum InkType : String {
        case pen

        case pencil

        case marker

        @available(iOS 17.0, macOS 14.0,
*)
        case monoline

        @available(iOS 17.0, macOS 14.0,
*)
        case fountainPen

        @available(iOS 17.0, macOS 14.0,
*)
        case watercolor

        @available(iOS 17.0, macOS 14.0,
*)
        case crayon

        /// The default width for an ink
type.
        public var defaultWidth: CGFloat
{ get }

        /// The valid width range for an
ink type.
        public var validWidthRange:
ClosedRange<CGFloat> { get }
```

```
        /// The PencilKit version
required to use this ink type.
        @available(iOS 17.0, macOS 14.0,
*)
        public var
requiredContentVersion: PKContentVersion
{ get }

        /// Creates a new instance with
the specified raw value.
        ///
        /// If there is no value of the
type that corresponds with the specified
raw
        /// value, this initializer
returns `nil`. For example:
        ///
        ///     enum PaperSize: String {
        ///         case A4, A5, Letter,
Legal
        ///     }
        ///
        ///     print(PaperSize(rawValue:
"Legal"))
        ///     // Prints
"Optional("PaperSize.Legal")"
        ///
        ///     print(PaperSize(rawValue:
"Tabloid"))
        ///     // Prints "nil"
        ///
        /// - Parameter rawValue: The raw
value to use for the new instance.
```

```
        public init?(rawValue: String)

        /// The raw type that can be used
to represent all values of the conforming
        /// type.
        ///
        /// Every distinct value of the
conforming type has a corresponding
unique
        /// value of the `RawValue` type,
but there may be values of the `RawValue`
        /// type that don't have a
corresponding value of the conforming
type.
        @available(iOS 13.0, macOS 11.0,
*)
        public typealias RawValue =
String

        /// The corresponding value of
the raw type.
        ///
        /// A new instance initialized
with `rawValue` will be equivalent to
this
        /// instance. For example:
        ///
        ///     enum PaperSize: String {
        ///         case A4, A5, Letter,
Legal
        ///     }
        ///
        ///     let selectedSize =
PaperSize.Letter
        ///
print(selectedSize.rawValue)
        ///     // Prints "Letter"
        ///
        ///     print(selectedSize ==
PaperSize(rawValue:
selectedSize.rawValue)!)
        ///     // Prints "true"
        public var rawValue: String { get
}
    }

    public var color: NSColor

    /// The base width of the ink.
    public var width: CGFloat

    /// The type of ink.
    public var inkType:
PKInkingTool.InkType

    /// The PencilKit version required to
use this inking tool.
    @available(iOS 17.0, macOS 14.0, *)
    public var requiredContentVersion:
PKContentVersion { get }

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (a:
PKInkingTool, b: PKInkingTool) -> Bool
}

@available(iOS 14.0, macOS 11.0, *)
extension PKInkingTool {

    /// Create a tool for the provided
ink.
    ///
    /// - parameter ink: The ink to use.
    /// - parameter width: The width of
stroke to create.
    public init(ink: PKInk, width:
CGFloat)

    /// The ink that this tool will
create strokes with.
    public var ink: PKInk { get }
}

@available(iOS 13.0, macOS 11.0, *)
extension PKInkingTool.InkType :
Equatable {
}
```

```
@available(iOS 13.0, macOS 11.0, *)
extension PKInkingTool.InkType : Hashable
{
}

@available(iOS 13.0, macOS 11.0, *)
extension PKInkingTool.InkType :
RawRepresentable {
}

@available(iOS 13.0, macOS 11.0, *)
public struct PKLassoTool : PKTool,
Equatable {

    /// Create a new lasso.
    public init()

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (a:
PKLassoTool, b: PKLassoTool) -> Bool
}
```

```
@available(iOS 14.0, macOS 11.0, *)
public struct PKStroke {

    public init(ink: PKInk, path:
PKStrokePath, transform:
CGAffineTransform = .identity, mask:
NSBezierPath? = nil)

    @available(iOS 16.0, macOS 13.0, *)
    public init(ink: PKInk, path:
PKStrokePath, transform:
CGAffineTransform = .identity, mask:
NSBezierPath? = nil, randomSeed: UInt32)

    /// The ink used to render this
stroke.
    public var ink: PKInk

    /// The affine transform of the
stroke when rendered.
    public var transform:
CGAffineTransform

    /// The B-spline path data that
describes this stroke.
    public var path: PKStrokePath

    public var mask: NSBezierPath?

    /// The bounds of the rendered
stroke.
    /// This includes the width & ink of
the stroke after the transform
```

```
    /// is applied.
    public var renderBounds: CGRect { get
}

    /// These are the parametric
parameter ranges of points in
`strokePath`
    /// that intersect the stroke's mask.
    public var maskedPathRanges:
[ClosedRange<CGFloat>] { get }

    /// The random seed for drawing
strokes that use randomized effects.
    @available(iOS 16.0, macOS 13.0, *)
    public var randomSeed: UInt32

    /// The PencilKit version required to
use this stroke.
    @available(iOS 17.0, macOS 14.0, *)
    public var requiredContentVersion:
PKContentVersion { get }
}

@available(iOS 14.0, macOS 11.0, *)
public struct PKStrokePath :
RandomAccessCollection {

    public init()

    /// Create a stroke path with the
given cubic B-spline control points.
    ///
    /// - parameter controlPoints: An
```

```
array of control points for a cubic B-
spline.
    /// - parameter creationDate: The
start time of this path.
    public init<T>(controlPoints: T,
creationDate: Date) where T : Sequence,
T.Element == PKStrokePoint

    public var creationDate: Date { get }

    /// The on-curve location for the
floating point [0, count-1]
`parametricValue` parameter.
    ///
    /// This has better performance than
`interpolatedPoint(at:
parametricValue).location`
    /// for when only the location is
required.
    public func interpolatedLocation(at
parametricValue: CGFloat) -> CGPoint

    /// The on-curve point for the
floating point [0, count-1]
`parametricValue` parameter.
    public func interpolatedPoint(at
parametricValue: CGFloat) ->
PKStrokePoint

    /// Returns an `InterpolatedSlice`
with a specific stride across a range of
this stroke data.
    ///
    /// - parameter range: The parametric
range to create a slice in.
    /// - parameter stride: The stride to
use between points
    /// - returns: A new slice that
strides across this stroke path with
`stride`.
    public func interpolatedPoints(in
range: ClosedRange<CGFloat>? = nil, by
stride:
PKStrokePath.InterpolatedSlice.Stride) ->
PKStrokePath.InterpolatedSlice

    /// Returns a parametric value on the
B-spline that is a specified offset from
the given parametric value.
    ///
    /// - parameter parametricValue: The
floating point [0, count-1] parametric
value.
    /// - parameter step: The distance to
offset `parametricValue`. `step` can be
positive or negative, distance or time.
    /// - returns: A parametric value
offset by `step` from `parametricValue`.
    public func parametricValue(_
parametricValue: CGFloat, offsetBy step:
PKStrokePath.InterpolatedSlice.Stride) ->
CGFloat

    public struct InterpolatedSlice {

        public enum Stride {
            case distance(CGFloat)

            case time(TimeInterval)

            case parametricStep(CGFloat)
        }
    }

    /// A type representing the
sequence's elements.
    public typealias Element =
PKStrokePoint

    /// A type that represents a position
in the collection.
    ///
    /// Valid indices consist of the
position of every element and a
    /// "past the end" position that's
not valid for use as a subscript
    /// argument.
    public typealias Index = Int

    /// The position of the first element
in a nonempty collection.
    ///
    /// If the collection is empty,
`startIndex` is equal to `endIndex`.
    public var startIndex: Int { get }

    /// The collection's "past the end"
position---that is, the position one
```

```
    /// greater than the last valid
subscript argument.
    ///
    /// When you need a range that
includes the last element of a
collection, use
    /// the half-open range operator
(`..<`) with `endIndex`. The `..<`
operator
    /// creates a range that doesn't
include the upper bound, so it's always
    /// safe to use with `endIndex`. For
example:
    ///
    ///     let numbers = [10, 20, 30,
40, 50]
    ///     if let index =
numbers.firstIndex(of: 30) {
    ///         print(numbers[index ..<
numbers.endIndex])
    ///     }
    ///     // Prints "[30, 40, 50]"
    ///
    /// If the collection is empty,
`endIndex` is equal to `startIndex`.
    public var endIndex: Int { get }

    /// Accesses the element at the
specified position.
    ///
    /// The following example accesses an
element of an array through its
    /// subscript to print its value:
```

```
    ///
    ///     var streets = ["Adams",
"Bryant", "Channing", "Douglas",
"Evarts"]
    ///     print(streets[1])
    ///     // Prints "Bryant"
    ///
    /// You can subscript a collection
with any valid index other than the
    /// collection's end index. The end
index refers to the position one past
    /// the last element of a collection,
so it doesn't correspond with an
    /// element.
    ///
    /// - Parameter position: The
position of the element to access.
`position`
    ///   must be a valid index of the
collection that is not equal to the
    ///   `endIndex` property.
    ///
    /// - Complexity: O(1)
    public subscript(index:
PKStrokePath.Index) -> PKStrokePoint {
get }

    /// A type that represents the
indices that are valid for subscripting
the
    /// collection, in ascending order.
    @available(iOS 14.0, macOS 11.0, *)
    public typealias Indices =
Range<PKStrokePath.Index>

    /// A type that provides the
collection's iteration interface and
    /// encapsulates its iteration state.
    ///
    /// By default, a collection conforms
to the `Sequence` protocol by
    /// supplying `IndexingIterator` as
its associated `Iterator`
    /// type.
    @available(iOS 14.0, macOS 11.0, *)
    public typealias Iterator =
IndexingIterator<PKStrokePath>

    /// A collection representing a
contiguous subrange of this collection's
    /// elements. The subsequence shares
indices with the original collection.
    ///
    /// The default subsequence type for
collections that don't define their own
    /// is `Slice`.
    @available(iOS 14.0, macOS 11.0, *)
    public typealias SubSequence =
Slice<PKStrokePath>
}

@available(iOS 14.0, macOS 11.0, *)
extension
PKStrokePath.InterpolatedSlice :
Sequence, IteratorProtocol {
    /// Advances to the next element and
returns it, or `nil` if no next element
    /// exists.
    ///
    /// Repeatedly calling this method
returns, in order, all the elements of
the
    /// underlying sequence. As soon as
the sequence has run out of elements, all
    /// subsequent calls return `nil`.
    ///
    /// You must not call this method if
any other copy of this iterator has been
    /// advanced with a call to its
`next()` method.
    ///
    /// The following example shows how
an iterator can be used explicitly to
    /// emulate a `for`-`in` loop. First,
retrieve a sequence's iterator, and
    /// then call the iterator's `next()`
method until it returns `nil`.
    ///
    ///     let numbers = [2, 3, 5, 7]
    ///     var numbersIterator =
numbers.makeIterator()
    ///
    ///     while let num =
numbersIterator.next() {
    ///         print(num)
    ///     }
    ///     // Prints "2"
    ///     // Prints "3"
    ///     // Prints "5"
    ///     // Prints "7"
    ///
    /// - Returns: The next element in
the underlying sequence, if a next
element
    ///   exists; otherwise, `nil`.
    public mutating func next() ->
PKStrokePoint?

    /// A type representing the
sequence's elements.
    @available(iOS 14.0, macOS 11.0, *)
    public typealias Element =
PKStrokePoint

    /// A type that provides the
sequence's iteration interface and
    /// encapsulates its iteration state.
    @available(iOS 14.0, macOS 11.0, *)
    public typealias Iterator =
PKStrokePath.InterpolatedSlice
}

@available(iOS 14.0, macOS 11.0, *)
public struct PKStrokePoint {

    public init(location: CGPoint,
timeOffset: TimeInterval, size: CGSize,
opacity: CGFloat, force: CGFloat,
azimuth: CGFloat, altitude: CGFloat)

    @available(iOS 17.0, macOS 14.0, *)
    public init(location: CGPoint,
timeOffset: TimeInterval, size: CGSize,
opacity: CGFloat, force: CGFloat,
azimuth: CGFloat, altitude: CGFloat,
secondaryScale: CGFloat)

    public var location: CGPoint { get }

    /// Time offset since the start of
the stroke in seconds.
    public var timeOffset: TimeInterval {
get }

    /// Size of the point.
    public var size: CGSize { get }

    /// Opacity of the point.
    public var opacity: CGFloat { get }

    /// Azimuth of the point in radians,
0.0–2π radians
    public var azimuth: CGFloat { get }

    /// Force used to create this point.
    public var force: CGFloat { get }

    /// Altitude used to create this
point in radians, 0.0–π/2 radians
    public var altitude: CGFloat { get }

    /// The scaling of the point for
secondary effects.
    @available(iOS 17.0, macOS 14.0, *)
    public var secondaryScale: CGFloat {
get }
}

@available(iOS 13.0, macOS 11.0, *)
public protocol PKTool {
}
```