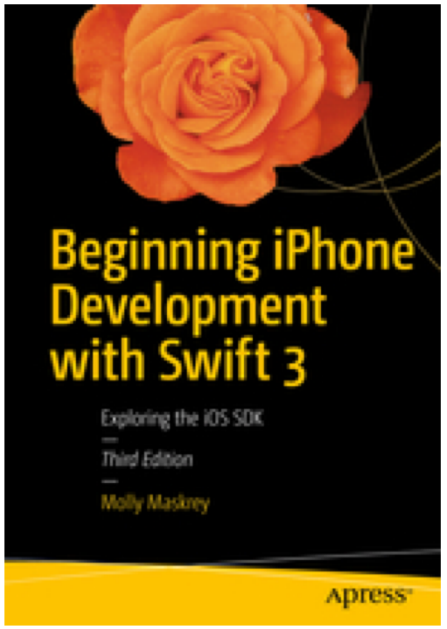
CENTENNIAL COLLEGE



Advanced iOS Development

Week 12

Maps

Maps

- Your app can imitate the **Maps** app, displaying a map interface and placing annotations and overlays on the map.
- The relevant classes are provided by the **Map Kit** framework.
- You'll need to import **MapKit**.
- The classes used to describe locations in terms of **latitude** and **longitude**, whose names start with "**CL**," come from the **Core Location** framework, but you won't need to import it explicitly if you're already importing the Map Kit framework.

Displaying a Map

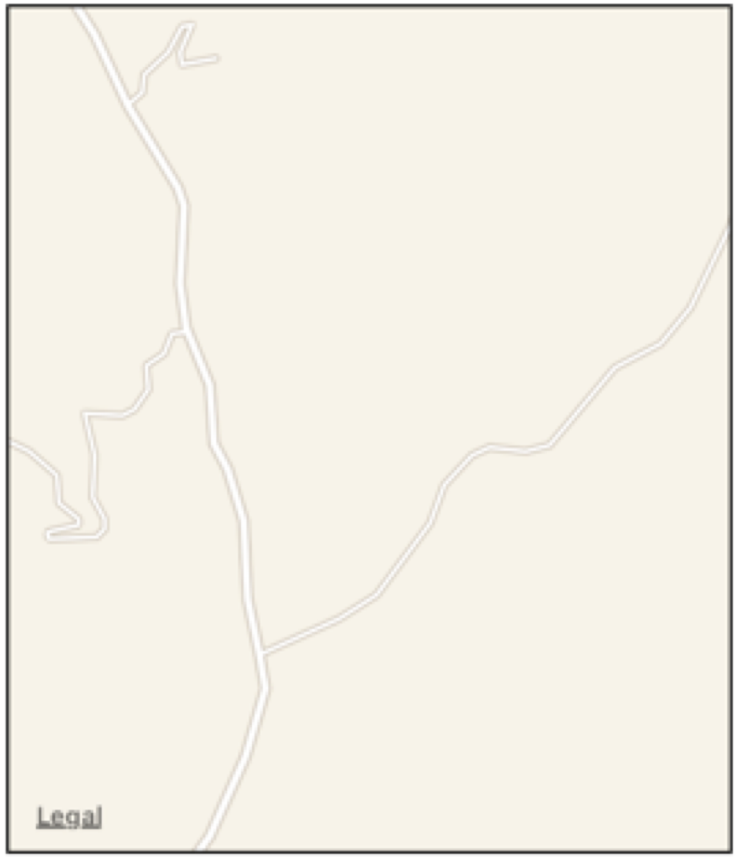
- A map is displayed through a UIView subclass, an **`MKMapView`**.
- You can instantiate an **`MKMapView`** from a **nib** or create one in code.
- A map has a type, which is usually one of the following (**`MKMapType`**):
  - **`.standard`**
  - **`.satellite`**
  - **`.hybrid`**
  - New in iOS 11, a further **MKMapType**, **`.mutedStandard`**, dims the map elements so that your additions to the map view stand out.

Displaying a Map (continued)

CENTENNIAL COLLEGE

- The area displayed on the map is its region, an **`MKCoordinateRegion`**.

- This is a **struct** comprising two things:
  - **`center`** - A **`CLLocationCoordinate2D`**. The **latitude** and **longitude** of the point at the center of the region.
  - span - An **`MKCoordinateSpan`**. The **quantity** of latitude and longitude embraced by the region (and hence the scale of the map).



Displaying a Map (continued)

- Convenience functions help you construct an **`MKCoordinateRegion`**.
- In this example, I'll initialize the display of an **`MKMapView`**

```
let loc = CLLocationCoordinate2DMake(34.927752,-120.217608)
let span = MKCoordinateSpanMake(0.015, 0.015)
let reg = MKCoordinateRegionMake(loc, span)
self.map.region = reg
```

- An **`MKCoordinateSpan`** is described in degrees of latitude and longitude. It may be, however, that what you know is the region's proposed dimensions in meters.

- To convert, call **MKCoordinateRegionMakeWithDistance**.

Displaying a Map (continued)

CENTENNIAL COLLEGE

- The ability to perform this conversion is important, because an `MKMapView` shows the world through a **Mercator projection**, where longitude lines are parallel and equidistant, and scale increases at higher latitudes.

- The **Mercator projection** is a **cylindrical map projection**. It became the standard map projection for nautical purposes because of its ability to represent lines of constant course, known as **rhumb** lines or **loxodromes**, as straight segments that conserve the angles with the meridians (Wikipedia).

Displaying a Map (continued)

- This is yet another way of displaying roughly the same region:

```
let loc = CLLocationCoordinate2DMake(34.927752,-120.217608)
let reg = MKCoordinateRegionMakeWithDistance(loc, 1200, 1200)
self.map.region = reg
```

Displaying a Map (continued)

- Yet another way of describing a map region is with an **`MKMapRect`**, a struct built up from **`MKMapPoint`** and **`MKMapSize`**.

- The earth has already been projected onto the map for us, and now we are describing a **rectangle** of that map, in terms of the units in which the map is drawn.

Displaying a Map (continued)

- The exact relationship between an **`MKMapPoint`** and the corresponding location coordinate is arbitrary and of no interest; what matters is that you can ask for the conversion, along with the ratio of points to meters (which will vary with latitude):
  - **`MKMapPointForCoordinate`**
  - **`MKCoordinateForMapPoint`**
  - **`MKMetersPerMapPointAtLatitude`**
  - **`MKMapPointsPerMeterAtLatitude`**
  - **`MKMetersBetweenMapPoints`**

Displaying a Map (continued)

- To determine what the map view is showing in **`MKMapRect`** terms, use its **`visibleMapRect`** property.

- Thus, this is another way of displaying approximately the same region:

```
let loc = CLLocationCoordinate2DMake(34.927752,-120.217608)
let pt = MKMapPointForCoordinate(loc)
let w = MKMapPointsPerMeterAtLatitude(loc.latitude) * 1200
self.map.visibleMapRect = MKMapRectMake(pt.x - w/2.0, pt.y - w/2.0, w, w)
```

Displaying a Map (continued)

- In none of those examples did we deal with the question of the actual dimensions of the map view itself.

- We simply threw a proposed region at the map view, and it decided how best to portray the corresponding area.

- Values you assign to the map views' region and **`visibleMapRect`** are unlikely to be the exact values it adopts, because the map view will optimize for display without distorting the map's scale.

- You can perform this same optimization in code by calling these methods:
  - **`regionThatFits(_:)`**
  - **`mapRectThatFits(_:)`**
  - **`mapRectThatFits(_:edgePadding:)`**

Displaying a Map (continued)

- By default, the user can **zoom** and **scroll** the map with the usual gestures; you can turn this off by setting the map view's **`isZoomEnabled`** and **`isScrollEnabled`** to false.
- Usually you will set them both to **true** or both to **false**.
- For further customization of an **`MKMapView`**'s response to touches, use a **`UIGestureRecognizer`**.
- You can change programmatically the region displayed, optionally with animation, by calling these methods:
  - **`setRegion(_:animated:)`**
  - **`setCenter(_:animated:)`**
  - **`setVisibleMapRect(_:animated:)`**
  - **`setVisibleMapRect(_:edgePadding:animated:)`**

Displaying a Map (continued)

- The map view's delegate (**`MKMapViewDelegate`**) is notified as the map loads and as the region changes (including changes triggered programmatically):
  - **`mapViewWillStartLoadingMap(_:)`**
  - **`mapViewDidFinishLoadingMap(_:)`**
  - **`mapViewDidFailLoadingMap(_:withError:)`**
  - **`mapView(_:regionWillChangeAnimated:)`**
  - **`mapView(_:regionDidChangeAnimated:)`**

- An **`MKMapView`** has Bool properties such as **`showsCompass`**, **`showsScale`**, and **`showsTraffic`**; set these to dictate whether the corresponding map components should be displayed.

Displaying a Map (continued)

CENTENNIAL COLLEGE

- New in iOS 11, the compass and the scale legend can be displayed as independent views, an **`MKCompassButton`** and an **`MKScaleView`**; if you use these, you'll probably want to set the corresponding Bool property to **false** so as not to get two compasses or scales.
- Both views are initialized with the map view as parameter, so that their display will reflect the rotation and zoom of the map.
- The **`MKCompassButton`**, like the internal compass, is a button; if the user taps it, the map is reoriented with north at the top.
- The visibility of these views is governed by properties (**`compassVisibility`** and **`scaleVisibility`**) whose value is one of these (**`MKFeatureVisibility`**):
  - **`.hidden`**
  - **`.visible`**
  - **`.adaptive`**
- The **`.adaptive`** behavior (the default) is that the compass is visible only if the map is rotated, and the scale legend is visible only if the map is zoomed.

Annotations

- An **annotation** is a ***marker*** associated with a location on a map.

- To make an annotation appear on a map, two objects are needed:
  - The **object** attached to the **`MKMapView`**

    The annotation itself is attached to the **`MKMapView`**. It is an instance of any class that adopts the **`MKAnnotation`** protocol, which specifies a coordinate, a title, and a subtitle for the annotation. You might have reason to define your own class to handle this task, or you can use the simple built-in **`MKPointAnnotation`** class. The annotation's coordinate is crucial; it says where on earth the annotation should be drawn. The **title** and **subtitle** are optional.

Annotations (continued)

- The object that **draws** the annotation

  An annotation is drawn by an **`MKAnnotationView`**, a **`UIView`** subclass. This can be extremely simple. In fact, even a nil **`MKAnnotationView`** might be perfectly satisfactory, because the runtime will then supply a view for you. In iOS 10 and before, this was a realistic rendering of a **physical pin**, red by default but configurable to any color, supplied by the built-in **`MKPinAnnotationView`** class. New in iOS 11, it is an **`MKMarkerAnnotationView`**, by default portraying a pin schematically in a circular red “balloon.”

Annotations (continued)

- Not only does an annotation require **two distinct objects**, but in fact those two objects **do not initially exist** together.

- An annotation object has no pointer to the annotation view object that will draw it. Rather, it is up to you to supply the annotation view object in real time, on demand. This architecture may sound confusing, but in fact it's a very clever way of reducing the amount of resources needed at any given moment.

- An annotation itself is merely a **lightweight object** that a map can always possess; the corresponding **annotation view** is a **heavyweight object** that is needed only so long as that annotation's coordinates are within the visible portion of the map.

Annotations (continued)

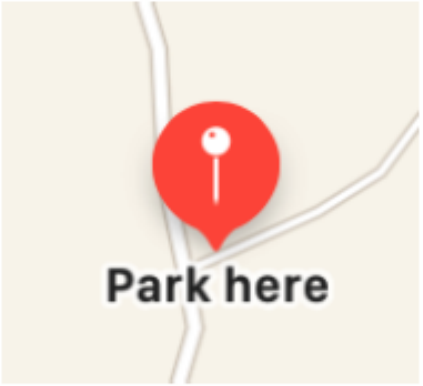
- Let's add the simplest possible annotation to our map.

- The point where the annotation is to go has been stored in an instance property (**`self.annloc`**):

```
let annloc = CLLocationCoordinate2DMake(34.923964,-120.219558)
```

- We create the annotation, configure it, and add it to the **`MKMapView`**:

```
let ann = MKPointAnnotation()
ann.coordinate = self.annloc
ann.title = "Park here"
ann.subtitle = "Fun awaits down the road!"
self.map.addAnnotation(ann)
```



Annotations (continued)

- By default, an MKMarkerAnnotationView displays its title below the annotation.

- This differs markedly from an **`MKPinAnnotationView`**, whose **title** and **subtitle** are displayed in a separate callout view that appears above the annotation view only when the annotation is selected (because the user taps it, or because you set the **`MKAnnotationView`**'s isSelected to true).

- A selected **`MKMarkerAnnotationView`** is drawn larger and displays the subtitle in addition to the title.

Customizing an `MKMarkerAnnotationView`

- **`MKMarkerAnnotationView`** has many customizable properties affecting its display.
- You can set the balloon color, as the view's **`markerTintColor`**. You can set the color used to tint the glyph portrayed inside the balloon, as the **`glyphTintColor`**.
- You can also change the balloon contents, overriding the default drawing of a pin.
- To do so, set either the **`glyphText`** (this should be at most one or two characters) or the **`glyphImage`**; in the latter case, use a **40 x 40** image, which will be sized down automatically to **20 x 20** when the view is not selected, or supply both a larger and a smaller image, the **`selectedGlyphImage`** and **`glyphImage`** respectively.
- The image is treated as a template image; setting the rendering mode to **`.alwaysOriginal`** has no effect.

Customizing an `MKMarkerAnnotationView` (Continued)

- In addition, you can govern the visibility of the title and subtitle, through the **`titleVisibility`** and **`subtitleVisibility`** properties.

- These are **`MKFeatureVisibility`** enums, where **`.adaptive`** is the default behavior that has already been described.

Customizing an `MKMarkerAnnotationView` (Continued)

- Doubtless you are now thinking: that's all very well, but what **`MKMarkerAnnotationView`** are we talking about? No such view appears in our code, so there is no object whose properties we can set!

- One way to access the annotation view is to give the map view a **delegate** and implement the **`MKMapViewDelegate`** method mapView(_:viewFor:) The second parameter is the **`MKAnnotation`** for which we are to supply a view.

- In our implementation of this method, we can **dequeue** an annotation view from the map view, passing in a string reuse identifier, similar to dequeuing a table cell from a table view. As we have taken no steps to the contrary, this will give us the default view, which in this case is an **`MKMarkerAnnotationView`**.

Customizing an `MKMarkerAnnotationView` (Continued)

- The notion of **view reuse** here is similar to the reuse of table view cells. The map may have a huge number of annotations, but it needs to display annotation views for only those annotations that are within its current region.

- Any extra annotation views that have been scrolled out of view can thus be reused and are held for us by the map view in a cache for exactly this purpose.

- The key to writing a minimal implementation of `mapView(_:viewFor:)` is to call this method:

  `dequeueReusableAnnotationView(withIdentifier:for:)`

  New in iOS 11. In the minimal case, pass as the identifier: the constant `MKMapViewDefaultAnnotationViewReuseIdentifier`. The second argument should be the annotation that arrived as the second parameter of the delegate method. This method will return an MKAnnotationView — the result is never nil as could happen in iOS 10 and before.

Customizing an `MKMarkerAnnotationView` (Continued)

- In this example, we check to see that my **`MKAnnotationView`** is indeed an **`MKMarkerAnnotationView`**, as expected:

```
func mapView(_ mapView: MKMapView,
    viewFor annotation: MKAnnotation) -> MKAnnotationView? {
        let id = MKMapViewDefaultAnnotationViewReuseIdentifier
        if let v = mapView.dequeueReusableAnnotationView(
            withIdentifier: id, for: annotation) as? MKMarkerAnnotationView {
                if let t = annotation.title, t == "Park here" {
                    v.titleVisibility = .visible
                    v.subtitleVisibility = .visible
                    v.markerTintColor = .green
                    v.glyphText = "!"
                    v.glyphTintColor = .black
                    return v
                }
        }
        return nil
}
```

Changing the Annotation View Class

- Instead of accepting the default **`MKMarkerAnnotationView`** as the class of our annotation view, we can substitute a different **`MKAnnotationView`** subclass. This might be our own **`MKMarkerAnnotationView`** subclass, or some other **`MKAnnotationView`** subclass, or **`MKAnnotationView`** itself.

- The way to do that in iOS 11 is to register our class with the map view, associating it with the reuse identifier, by calling **`register(_:forAnnotationViewWithReuseIdentifier:)`** beforehand.

Changing the Annotation View Class (continued)

- To illustrate, we'll use **`MKAnnotationView`** itself as our annotation view class.

- We won't get the default drawing of a balloon and a pin, because we're not using **`MKMarkerAnnotationView`** any longer; instead, we'll set the **`MKAnnotationView`**'s **`image`** property directly.

- We also won't get the **title** and **subtitle** drawn beneath the image; instead, we'll set the annotation view's **`canShowCallout`** to true, and the title and subtitle will appear in the callout when the annotation view is selected.

Changing the Annotation View Class (continued)

- So, assume that I have an identifier declared as an instance property:

```
let bikeid = "bike"
```

- And assume that I've registered **`MKAnnotationView`** as the class that goes with that identifier:

```
self.map.register(MKAnnotationView.self,
    forAnnotationViewWithReuseIdentifier: self.bikeid)
```

Changing the Annotation View Class (continued)

- Then my implementation of **`mapView(_:viewFor:)`** might look like this:


```
func mapView(_ mapView: MKMapView,
    viewFor annotation: MKAnnotation) -> MKAnnotationView? {
        let v = mapView.dequeueReusableAnnotationView(
            withIdentifier: self.bikeid, for: annotation)
        if let t = annotation.title, t == "Park here" {
            v.image = UIImage(named:"clipartdirtbike.gif")
            v.bounds.size.height /= 3.0
            v.bounds.size.width /= 3.0
            v.centerOffset = CGPoint(0,-20)
            v.canShowCallout = true

                return v
            }
            return nil
    }
```

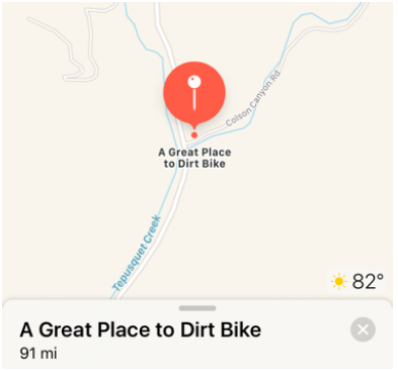
Communicating with the Maps App

- Your app can communicate with the Maps app.
- For example, instead of displaying a point of interest in a map view in our own app, we can ask the **Maps app** to display it.
- The user could then bookmark or share the location. The channel of communication between your app and the Maps app is the `MKMapItem` class.

Communicating with the Maps App (continued)

- Here, we'll ask the Maps app to display the same point marked by the annotation in our earlier examples, on a standard map portraying the same region of the earth that our map view is currently displaying:

```
let p = MKPlacemark(coordinate:self.annloc, addressDictionary:nil)
let mi = MKMapItem(placemark: p)
mi.name = "A Great Place to Dirt Bike" // label to appear in Maps app
mi.openInMaps(launchOptions:[
    MKLaunchOptionsMapTypeKey: MKMapType.standard.rawValue,
    MKLaunchOptionsMapCenterKey: self.map.region.center,
    MKLaunchOptionsMapSpanKey: self.map.region.span
])
```



Communicating with the Maps App (continued)

- If you start with an **`MKMapItem`** returned by the class method **`mapItemForCurrentLocation`**, you're asking the Maps app to display the device's current location.

- This call doesn't attempt to determine the device's location, nor does it contain any location information; it merely generates an **`MKMapItem`** which, when sent to the Maps app, will cause it to attempt to determine (and display) the device's location:

```
let mi = MKMapItem.forCurrentLocation()
mi.openInMaps(launchOptions:[
    MKLaunchOptionsMapTypeKey: MKMapType.standard.rawValue
])
```