GESTURES

GESTURES

- Gestures offer a simplified way to react on events like tapping or swiping.
- Are optimized on performance
- Are the preferred way to handle gestures

GESTURES ON A VIEW

Tapping	**UITapGestureRecognizer**
Pinching (=zooming) in and out	UIPinchGestureRecognizer
Panning or dragging	UIPanGestureRecognizer
Swiping	UISwipeGestureRecognizer
Rotating	UIRotationGestureRecognizer
Long press	UILongPressGestureRecognizer

ADD A GESTURES TO A VIEW

```
override func viewDidLoad() {
    super.viewDidLoad()

    //create a tab gesture for a double click
    let gesture : UITapGestureRecognizer = UITapGestureRecognizer();
    gesture.numberOfTapsRequired = 2;

    gesture.delegate = self;

    //note the : at the end. We want to create a handler with parameters
    gesture.addTarget(self, action: "handleGesture:");

    self.view.addGestureRecognizer(gesture);
```

ADD A GESTURES HANDLER

```
//do whatever you like in the handler
func handleGesture(gestureRecognizer: UIGestureRecognizer) {
    print ("gesture active");
}
```

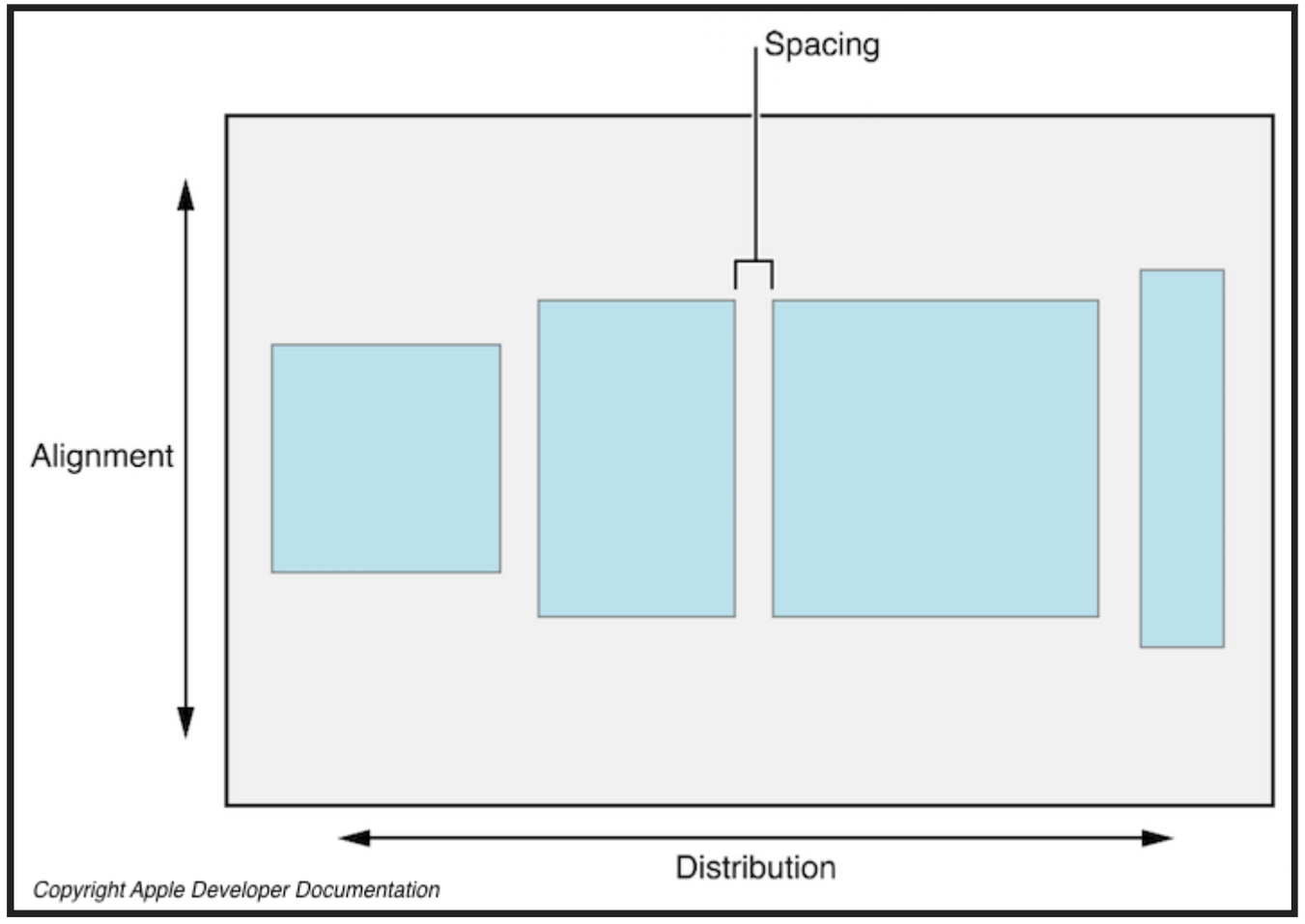
EXERCISE

Create your own project with a long press gesture.

UISTACKVIEW

UISTACKVIEW BASICS

- Introduced in 2015, UIStackView can be used to generate layouts without autolayout easily.
- A UIStackView can either be layouted horizontally or vertically (rows or columns).
- You can pin a stackview on two borders (ie. top left corners), on three or four borders.
- The attributes distribution, alignment and spacing control how the components within the view are rendered.


Spacing
Alignment
Distribution
Copyright Apple Developer Documentation

EXERCISE

Create a new project and add three buttons vertically. Stack them. Use autolayout to pin the newly created stack view on the top, left and bottom edge of the superview. Use Distribution -> Equal spacing to equally space the buttons vertically.

PERSISTENCE

KEY VALUE STORAGE I

- The most simple storage, can be used with two lines of code
- Can be synchronized between apps using iCloud
- Is sufficient for many apps

KEY VALUE STORAGE II

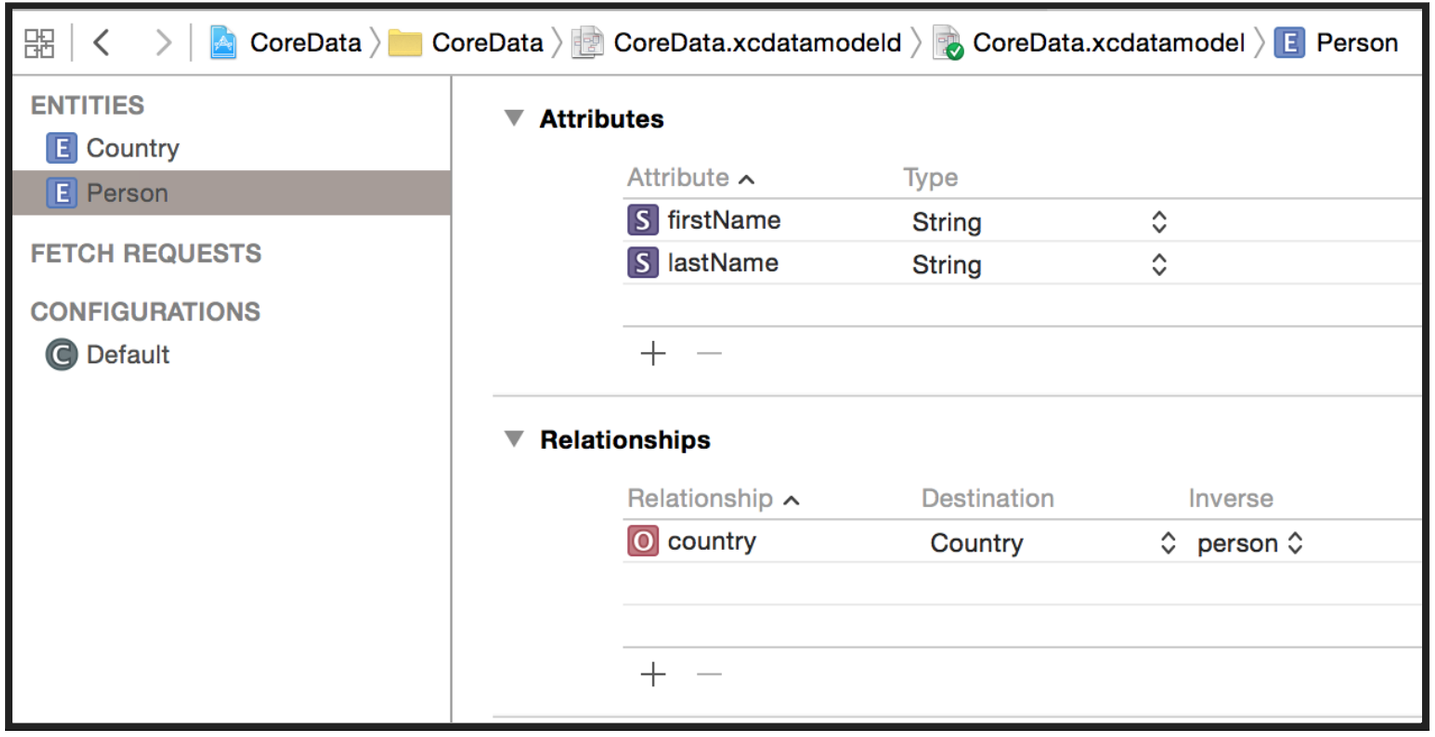
```
//create the key value entry
NSUserDefaults.standardUserDefaults()
    .setInteger(50, forKey: "MaximumNumberOfConnections");

//and save the entry
NSUserDefaults.standardUserDefaults().synchronize();

//reading a saved entry is also straightforward
print(NSUserDefaults.standardUserDefaults()
    .integerForKey("MaximumNumberOfConnections"));
```

CORE DATA STORAGE

- Core Data is not a relational database but you also have entities and relations.
- There is a bit of boilerplate code when using Core Data. Xcode will generate this for you when you select "Use CoreData" during project generation.
- The main part of CoreData is a .xcdatamodelId file where all entities are managed.

SETTING UP CORE DATA STORAGE I

Define your entities in the .xcdatamodelId file (ie Person and Country).



SETTING UP CORE DATA STORAGE II

- Note that you can tune all attributes and relationships on the right panel.
- When the model is finished, you create classes by clicking on "Editor->Create NSManagedObject Subclass".

SETTING UP CORE DATA STORAGE III

```
//in order to access the storage, we need a managedObjectContext
//you can create this as a member variable whenever you need access
  let managedObjectContext =
        (UIApplication.sharedApplication().delegate as! AppDelegate).managedObjectContext

//now create a new addresss
let newPerson = NSEntityDescription.insertNewObjectForEntityForName("Person",
        inManagedObjectContext: managedObjectContext) as! Person

//add some values
newPerson.firstName = "Elvis"
newPerson.lastName = "Presley"

//this entity is now stored in our context (but not persisted). However, during
//a query, this object will be returned!
```

SETTING UP CORE DATA STORAGE IV

```
//and save
//attention: This will save all newly created entities in this context!
do {
        try managedObjectContext.save();
} catch _ {
        print("exception when saving our person");
}
```

QUERYING CORE DATA STORAGE I

```
//for reading our storage, do a fetch request
let fetchRequest = NSFetchRequest(entityName: "Person")

//and read
if let fetchResults = try self.managedObjectContext.executeFetchRequest(fetchRequest)
                as? [Person] {

        //we now have all addresses in our fetchResults array
        for address in fetchResults {
                print("firstName: " + address.firstName! +
                      ", lastName: " + address.lastName!);
        }
}
```

QUERYING CORE DATA STORAGE II

```
//You can define a sort order
let fetchRequest = NSFetchRequest(entityName: "Person")

//create a sort descriptor with key and ascending/descending
let sortDescriptor = NSSortDescriptor(key: "lastName", ascending: true)

//add the sort descriptor to your request
fetchRequest.sortDescriptors = [sortDescriptor]

//if we execute the fetch request now, we retrieve our addresses ordered by the
//first name.
```

QUERYING CORE DATA STORAGE III

```
//Filtering data can be done on field level
let predicate = NSPredicate(format: "lastName == %@", "Hemingway")

// Set the predicate on the fetch request
fetchRequest.predicate = predicate

//if we execute the fetch request now, we retrieve only the addresses where lastName
//is exactly "Hemingway"
```

QUERYING CORE DATA STORAGE IV

```
//you can also vary the query (lastName starts with a P)
let predicate = NSPredicate(format: "lastName like ", "P*")


//lastName starts with an H or firstName ends with a t
let predicate = NSPredicate(format:
        "(lastName like %@) OR (firstName like %@)", "S*", "*t")
```

DELETING ITEMS IN CORE DATA STORAGE

```
if let fetchResults = try self.managedObjectContext.executeFetchRequest(fetchRequest) as?
    for address in fetchResults {

        //delete the object. This method awaits an NSManagedObject as parameter
        self.managedObjectContext.deleteObject(address)

    }

    //make sure that the context is saved
    try self.managedObjectContext.save();
}
```

UPDATES ON NSMANAGEDOBJECTS

```
//get an NSManagedObject (country) by fetching it from the context
let country = try getCountryWithName("UK")

//change the name
country!.name = "United Kingdom"

//this new name is now automatically propagated to all relations
let person = try getPersonWithFirstName("Charles")
```

ADD MIGRATION OPTIONS FOR CORE DATA CHANGES

```
//if you change the CoreData object, the application will crash during next startup.
//You need to add migration options to the generated stub
let mOptions = [NSMigratePersistentStoresAutomaticallyOption: true,
    NSInferMappingModelAutomaticallyOption: true]

try coordinator.addPersistentStoreWithType(NSSQLiteStoreType, configuration: nil, URL: url
```

FURTHER CORE DATA FEATURES

- You can use Core Data as an in-memory database.
- You don't need to use searching (fetching). You can use Core Data and have all objects in your memory. However, updates are still automatically propagated (see previous slides).
- You can be informed when an update occurs (trigger).

MISSING CORE DATA FEATURES

- You need to load your data into memory before you can operate on them (No deletion without loading).
- No transactions!
- No multi-threading
- No multi-user
- No data constraints like "unique"