iOS Dev Accelerator
Week 6 Day 4

- CoreData/Testing in Swift
- Version Migration
- iCloud
- Hash Tables

Testing with CoreData

Writing tests in swift

- Writing tests in swift still follows the same principles we covered yesterday (XCTest, F.I.R.S.T), but there a few gotchas
- Swift is namepaced, which is why we dont have to #import <ClassName>.h in our swift files. Everything in your Swift app's application target, or module, knows about everything else.
- This is big improvement over Objective-C, but for writing tests it adds an extra hurdle
- You will need to do 2 things to write tests in swift projects

Writing tests in swift

1. At the top of your test file, import your Swift application's module. This will look like:
    - import Name_of_your_project
        - No quotes or .swift necessary. It probably wont autofill which sucks. If you have spaces in your project name, use underscores.
2. Every custom class you are going to test, you must add to the test target as well.

Demo

Swift and CoreData

Swift and CoreData

- Swift and CoreData works pretty much exactly like Objective-C and CoreData.
- Same stack, same MOM, same methods
- The only weirdness comes when generating your custom subclasses with Xcode.

Swift and CoreData

- After generating your subclasses using the auto generate feature in Xcode, you must modify the ‘Class’ entry in the attribute inspector of each entity in your MOM file.
- You simply need to add the name of your project and then a dot, and then the name of the entity.
- So if your project is called CoreDataHotel and your entity is called Hotel, it should say CoreDataHotel.Hotel.
- If you forget to do this, your app will crash when you try to use a custom subclass object and print this error message to the console:
- “Unable to load class named ‘ClassName’ for entity ‘EntityName’, using default NSManagedObject instead.

Demo

Versioning & Migration

Migrating with Core Data

- Its pretty common for your Managed Object Model to change during the development of your app
- Especially between releases, as you are adding or removing new features
- So the question is, how do we account for user's data when they update their app, and the MOM has changed? Their old entities, attributes, and relationships probably wont match the new MOM layout, which is a big problem. How can we properly accommodate our users?
- With Migration.

Migrating with Core Data

- A rule of thumb: Anytime you make changes to your data model, you need to migrate
- However, certain situations can avoid migrations all together. For example, if you are using core data as an offline cache, you can simply clear the old cache out and start new.
- When migration is needed, you need to **a)create a new version of the data model, and b)provide a migration path**

Making the decision

- When you first add a store to your persistent store coordinator, Core Data does a few things under the hood:
- Core Data compares the store’s model version with the coordinators model version.
- If the versions don't match, Core Data will perform a migration (if enabled)

Starting the process

- When the migration process starts, Core Data looks at both the original data model and the destination/new model.
- It use these two models to create a mapping model for the migration.
- Core Data uses this mapping model to convert data in the original store so it can be used in the new model

Under the hood

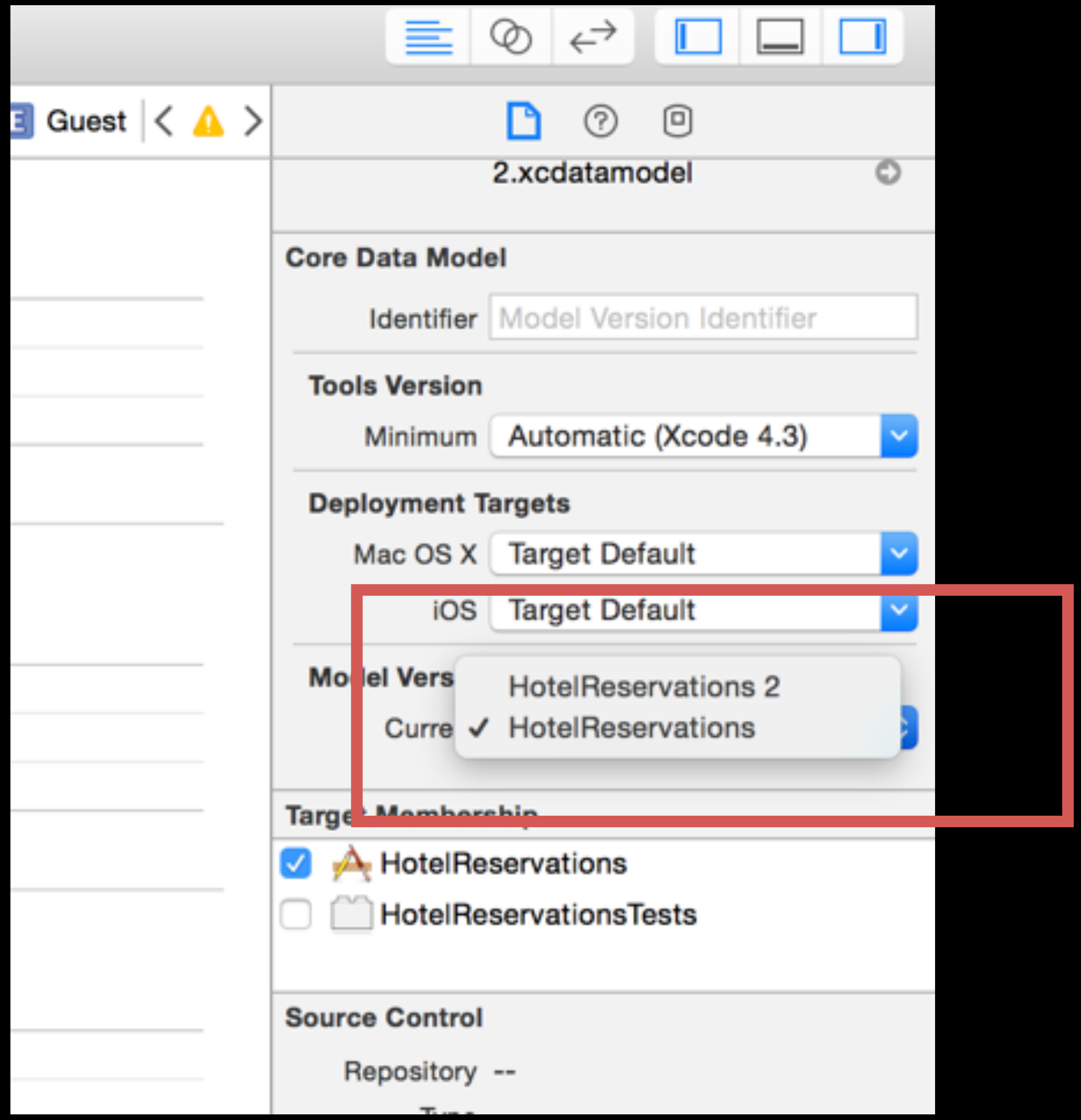
- Under the hood, the migration has 3 steps:
    1. Core Data copies all of the objects from the old data store to the next store
    2. Core Data connects all the objects according to the relationships
    3. Core Data enforces any data validation in the destination model
- If something goes wrong, core data reverts to the old data store. The old data is never deleted until a successful migration is achieved.

The different types of a Core Data Migration

- Officially, Apple only recognizes two types of migrations: lightweight and not lightweight migrations
- But really there is a few different types of migrations, ordered from simplest to most complex:
- **Lightweight migration**: least amount of work involved for you. Its as simple as enabling a couple of flags when you setup your stack. But there are limitations.
- **Manual migration:** More work for you. You have to manually specify how the old data will map to the new data model. GUI tools are provided in Xcode, similar to how you setup your MOM
- **Custom Manual Migration:** You use the GUI mapping model from manual migration, but also add custom code to specific any custom transformation logic.
- **Fully Manual Migration:** Custom version detection logic and custom handling of the migration process

Lightweight Migration

- Lightweight Migration is pretty straight forward process:
  1. With your MOM file open, click Editor>Add Model Version and give your new model a name
  2. Change the version of your MOM to the new version:
  3. Make your changes to your MOM
  4. Generate new subclasses
  5. Modify your persistent store to contain special options to enable automatic migration



2.xcdatamodel

Core Data Model

Identifier: Model Version Identifier

Tools Version

Minimum: Automatic (Xcode 4.3)

Deployment Targets

Mac OS X: Target Default

iOS: Target Default

Model Vers

Curre ✓ HotelReservations

HotelReservations 2

Target Membership

✓ HotelReservations

HotelReservationsTests

Source Control

Repository --

Enabling Automatic Migration

- In order for Core Data to automatically migrate between your MOM versions, you need to include a specific set of options when adding your persistent store to your PSC
- The options parameter of addPersistentStoreWithType:configuration:URL:options:error: takes in a dictionary of options.
- The two key-value pairings you need are:
    - NSMigratePersistentStoreAutomaticallyOption : true (enables migration in general)
    - NSInferMappingModelAutomaticallyOption : true (tells core data to automatically merge if its capable, set to false if you are doing a manual merge)

Lightweight Limitations

- Lightweight Migrations have some limitations. They can do pretty much any obvious migration pattern:
  - Deleting entities, attributes, relationships
  - Renaming entities, attributes, relationships
  - Adding new attributes
  - Changing optional attributes to required or vice versa
  - Changing entity hierarchy
  - changing a relationship from one to many
  - changing a to many relationship to ordered or unordered
- As you can see, lightweight migrations are pretty powerful! They should cover 90% of your migration scenarios. You should always strive to keep your migrations lightweight.

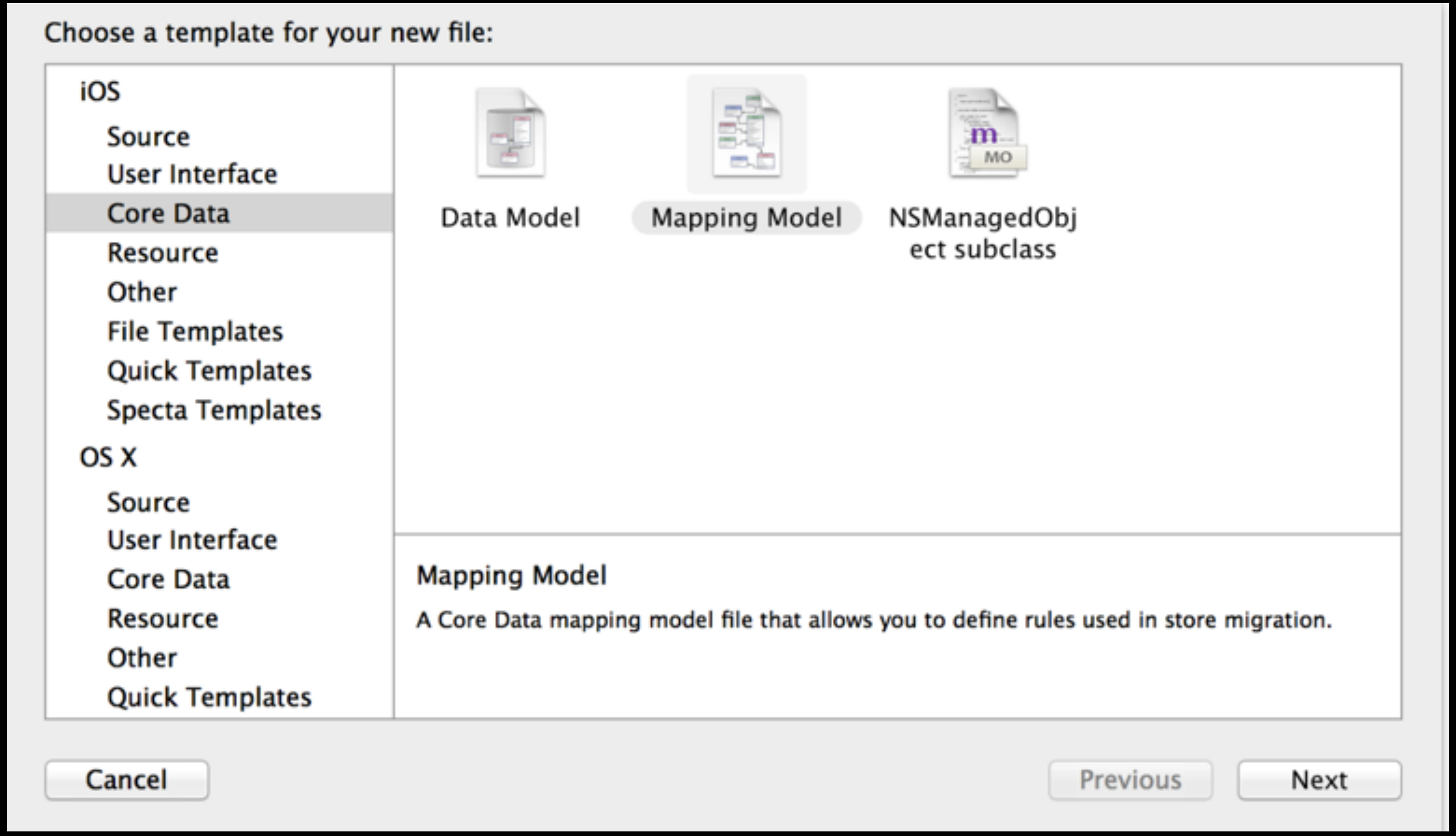
Demo

Manual Migration

- Manual Migrations are more work for you, the developer
- Thankfully Xcode has a GUI style editor for manual migrations, similar to the editor you use to setup your MOM
- After making a new version of your MOM with your scheme changes, you need to create a mapping file

Mapping File

- To create a mapping file, go to File> New File>iOS>CoreData>Mapping Model
- Select the data model you are mapping from, as the source, and to, as the destination



Mapping Model

A Core Data mapping model file that allows you to define rules used in store migration.

- After getting your mapping file setup, the last step is to modify your core data stack setup options by setting the key NSInferMappingModelAutomaticallyOption to false

Migrating multiple versions

- If your app has multiple migrations, you will need to account for the fact that some users may skip updates and attempt to migrate from versions that aren't right after each other.
- For example, a user may download your app in its initial launch, and then wait 6 months to update, which means they missed an update entirely. Their data would then need to be migrated from v1 to v3, skipping v2.
- What you have to do is heavily modify your core data stack to check which version the user is currently on, and perform each sequential migration recursively until they are at the latest version. (this sucks but is doable)

iCloud

iCloud

- “iCloud is a cloud service that gives your users a consistent and seamless experience across all of their iCloud-enabled devices”
- iCloud works with things called ‘**ubiquity** containers’, which are special folders that your app uses to store data in the cloud.
- Whenever you make a change in your ubiquity container, the system uploads the changes to the cloud.
- iCloud is closely integrated with CoreData to help persist your manage objects to the cloud. This is another incentive to use core data to manage your model layer!

Enabling iCloud in your app

- To use iCloud in your app, you have to enable iCloud support in your app and in your project.
- Navigate to your app's target in the project settings in Xcode, and select the capabilities pane. Switch the iCloud option on and follow the configuration prompts.
- Put checks next to iCloud Documents and CloudKit
- Once that is setup you just need to sign into iCloud on your simulator. Its easier to use your own account, but you can setup a test iCloud account on iTunes Connect.

Enabling iCloud in your Core Data Stack

- iOS 8 makes enabling iCloud in your core stack is as simple as setting up your persistent store with a dictionary containing a few special options:
    - NSPersistentStoreUbiquitousContentNameKey - option to specify that a persistent store has a given name in ubiquity
    - NSMigratePersistentStoresAutomaticallyOption - key to automatically attempt to migrate versioned stores
    - NSInterMappingModelAutomaticallyOption - key to attempt to create the mapping model automatically

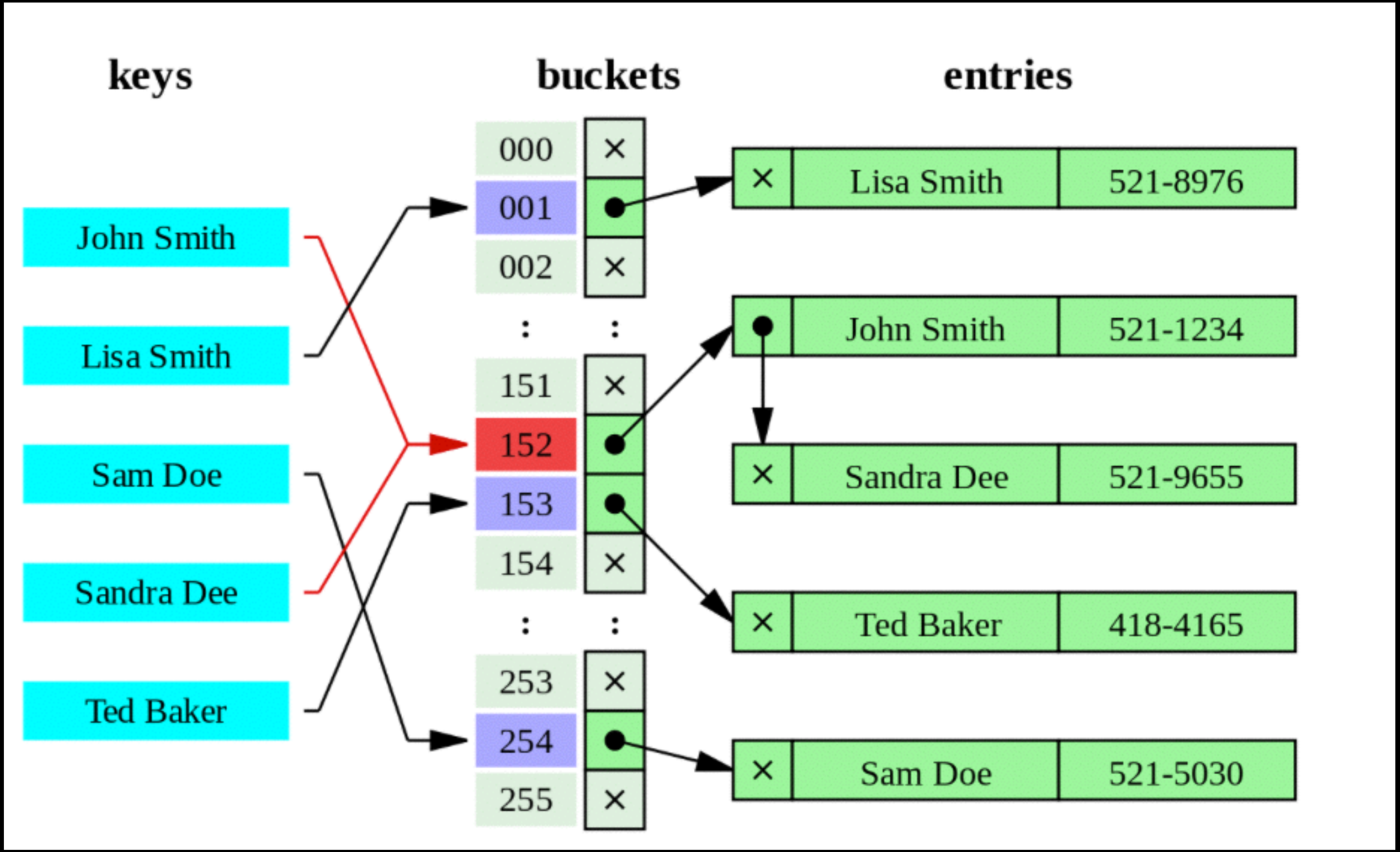
Demo

Hash Tables

Hash Tables

- A hash table is a data structure that can map keys to values. A dictionary is a hash table.
- The key component to a hash table is a hash function. Given a key, the hash function computes an index to store the values in a backing array.
- On average, hash tables have an O(1) constant time look up. It’s amazing
- Ideally the hash function will assign each key to unique index, but usually in practice you have collisions. In this case we use our old friend linked list to help us out.

Hash Tables


keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
151
152
153
154
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

Hash Tables Workflow

- This is what happens when you add a value to a hash table/dictionary:
    1. The key passed in with the value to add is run through the hash function, which returns an index number
    2. The hash table creates a new bucket, with the value and key
    3. The hash table goes to the index of its backing array which matches the index returned from the hash function, and inserts the bucket at the head of the linked list of buckets.

Hash Tables Collisions

- Whenever you have more than one bucket in the linked list at each index of the backing array, that is considered a collision.
- Collisions are bad, because it takes the lookup time of a Hash Table from O(1) constant time to O(n) linear time. :(
- It's linear because you have to search through the linked list of buckets to find the exact key value pairing you were looking for, and worst case its at the end of the linked list.
- Better hash functions improve the rate of collisions, but they are never completely eliminated.

Hash Tables Workflow

- This is what happens when you try to retrieve a value for a key from a hash table:
  1. The key passed in with the value to add is run through the hash function, which returns an index number
  2. The hash table traverses through the linked list of buckets at that index of its backing array, and returns the value if it finds a bucket with the given key.
  3. If you have no collisions, this is constant time, if you have collisions its linear time.

Modulus Operator

- We are going to use the modulus operator in our simple hash function.
- The modulus operator finds the remainder of division of one number by another.
- so 10 % 3 is 1 because 3 goes into 10 3 times, and then a 1 is left.
- modulus is useful in our hash table implementation because our hash function needs a way to make sure the index it produces is always in the range of our array size.

Demo