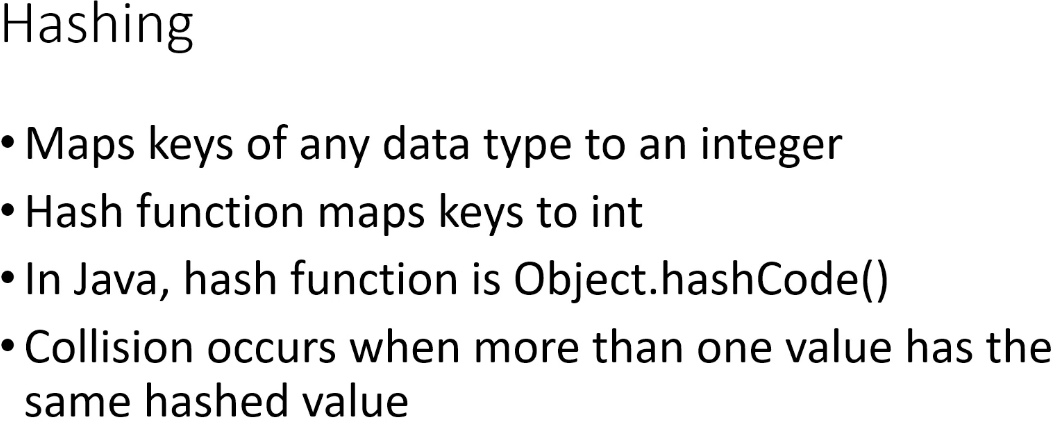
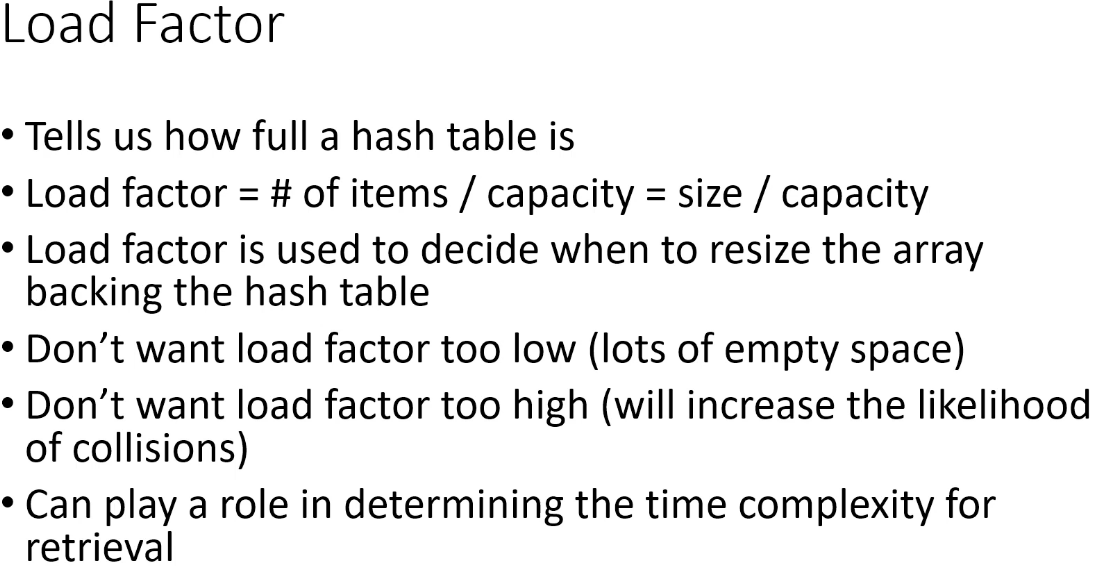
**Introduction to Hashtables**  
\* **Hash Table** is an **Abstract Data Type** just like Stacks and Queues.  
\* You may have worked with a Hashtable without realizing that you were working with a Hashtable.

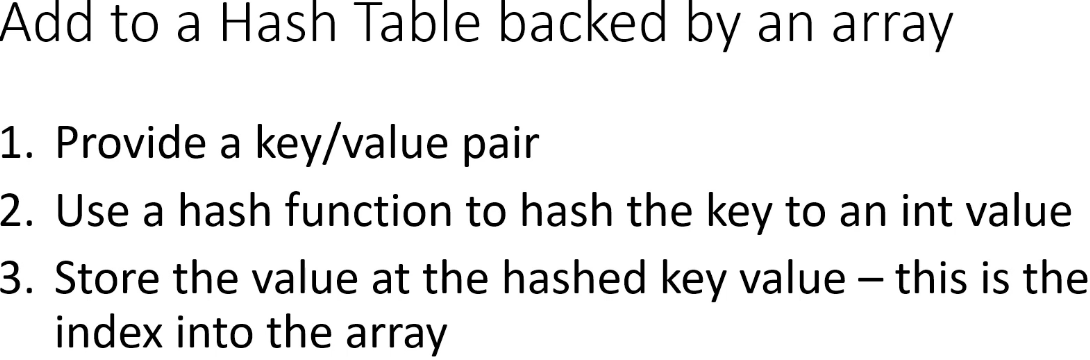
**Hashtables (Theory)**  
=> **Abstract Data Type** - **doesn’t dictate how you store the data**.  
=> **You can back a Hashtable with whatever you want**.  
=> Provides access to data using keys.  
=> Key doesn’t have to be an integer.  
=> **Consists of key/value pairs** - data types don’t have to match.  
=> **Optimized for retrieval** (**when you know the key**)

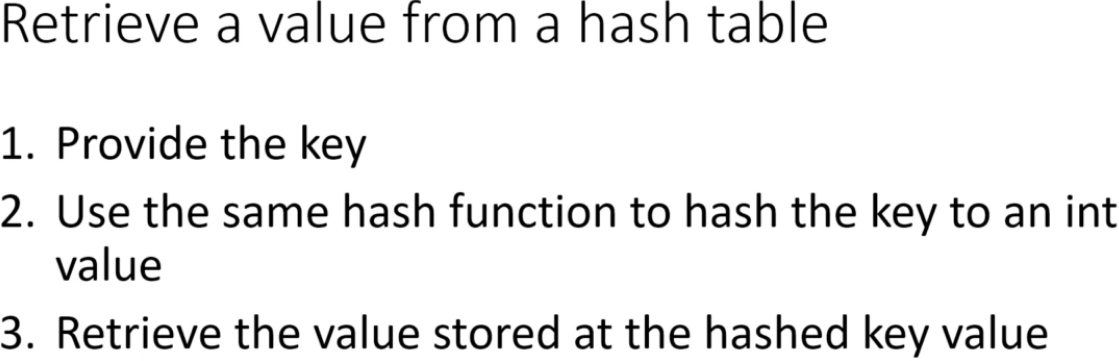
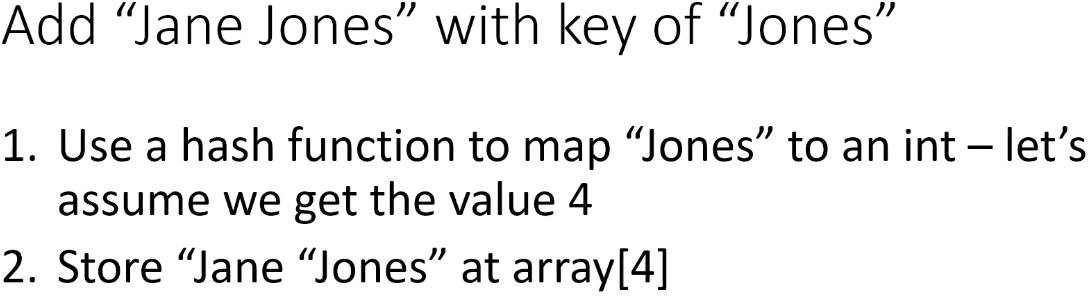
=> **Associative array is one type of Hash Table**.

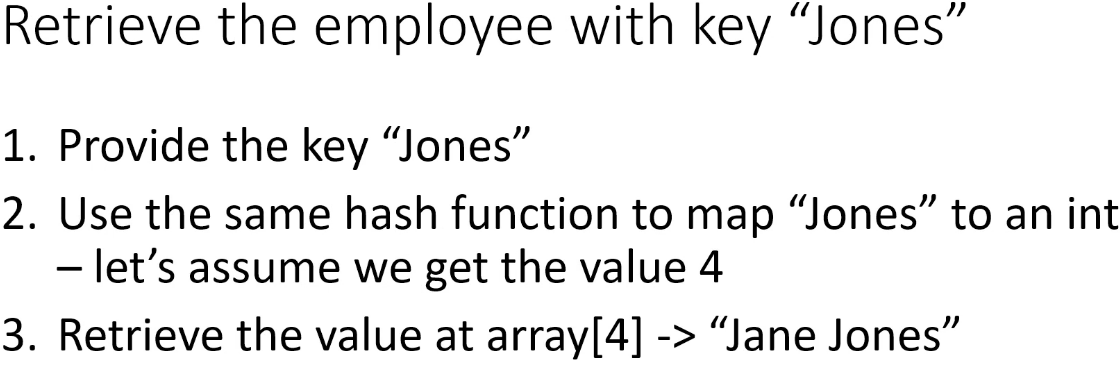
\* When you add an item, you provide the key and the value.  
\* When you retrieve an item, you provide the key.  
\* **It can retrieve the value really, really quickly**.  
\* **If you don’t know the key, the retrieval is going to be slow**.  
\* When we’re talking about arrays, we could see the array index as the key and the value as the element in the array.  
\* But, for Hash Tables, the **key doesn’t have to be an integer**.  
\* **Strings are a common data type that’s used for keys**.  
\* Hash Tables provide direct access to values in the table, using keys.  
\* **Other words for a** **Hash Table**:   
**Dictionaries  
Maps  
Lookup Tables  
Associative Arrays**  


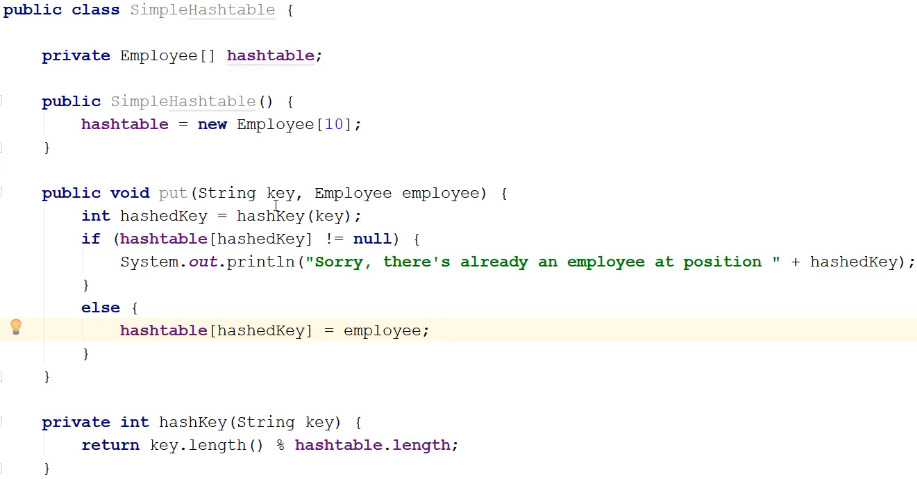
=> **Under the covers, those keys are being converted to integers**.  
=> One common way of backing a Hash Table is to use an array.  
**Hashing** => **convert the keys of any data type to integers**, you **hash the key**.  
\* Hash function maps keys to int.  
**Object.hashCode()** => in Java, it is **often overriden**.  
\* It’s possible that the hashing method may produce the same integer for more than 1 value, when that happens, it’s known as a **collision**.  
\* For example if we have more than 1 employee with the last name Jones and we use that as the key, they’re going to get the same result.  
=> **There are strategies for dealing with collisions**.  
\* Some examples of keys/values would be: product numbers and products, employee IDs and employee objects, usernames and profiles - when you create an account at a website and you provide a username, that, under the covers, may be getting hashed to an integer and that integer is used to look up your profile.

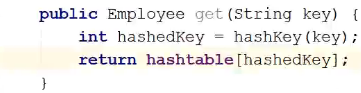
  
\* **One important value for Hash Tables is**:  
**Load Factor** - tells us how full a Hash Table is.  
\* Let’s say we’re backing the Hash Table with an array, the Load Factor would tell us how full the array is and we can get that by dividing the number of items by the capacity, which is essentially the **size / capacity**.  
\* So if we had an array of 10 backing our Hash Table and we had 5 employees currently in the Hash Table, the Load Factor would be 5 / 10 = 0.5. That tells us that our array is half full.  
=> The Load Factor is used to decide when to resize the array that’s backing the Hash Table for example.  
=> It’s kind of a balancing act, we don’t want the Load Factor to be too low because that means there’s lots of empty space. But we don’t want it to be too high because that increases the likelihood of collisions. If most of the array is full and you keep adding items, you are going to get collisions eventually because you’re basically Hashing the keys to a range of values, essentially the valid indices in the array and so if your array is already pretty full and you’re going to be adding lots more items, you’re going to start to get collisions.  
\* **Can play a role in determining the Time Complexity for retriaval**.

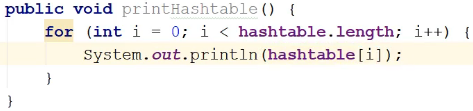


  
\* **If by any chance we’d had collisions, we’d have to search through all the items, in that case our array would probably contain a Linked List or each item in the array would be another array**.  


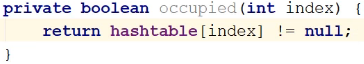
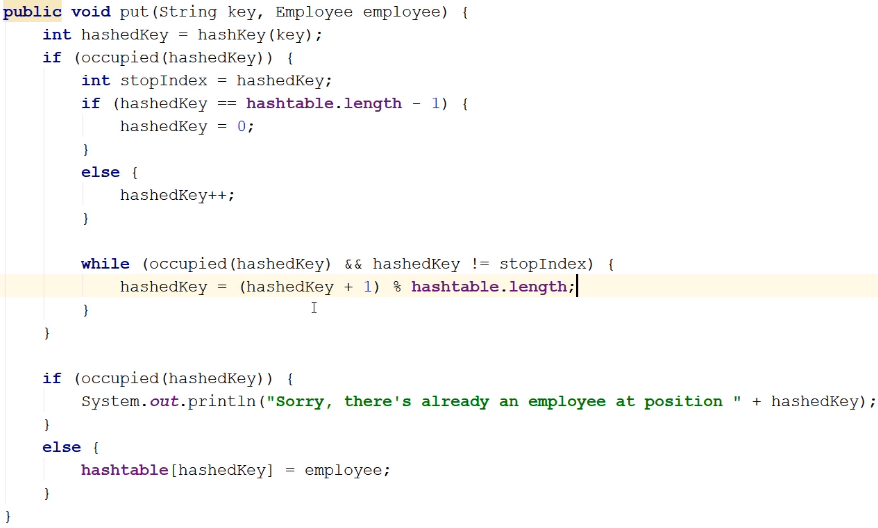


**Hashtables (Array Implementation)**  
\* **Hash Tables exist to provide speedy retrieval of data**.  
\* We accomplish that by taking the key-value and we use a hash function to map the key to an integer and then we use that integer to retrieve the item.  
**array** => One data structure that would land itself really well to this is array.  
\* Because when we hash the key to an integer, we can use that integer as the index into the array.  
\* It’s important to understand that **the key and the hashed value are not the same thing**.  
\* The **hashed value is always an integer**, the key can be anything.  
\* This is a super simple hash function which gives us indices 0 - 9.  
\* The hashing function is going to play a huge role in how quick your retrievals are, if you have a hashing function that ends up hashing every single key to the same value, your retrievals are going to be pretty slow because all of your values are going to go into the same index in the array and you’re going to have to search through all your values to find the one you’re looking for.  
\* If you have a hashing function that evenly distributes the values in the array, then your retrievals are going to be better.  
\* The ideal case is obviously that you only ever have 1 value per key.  
\* **In the real world, we probably use the ID as the key because they’re probably going to be unique**. **But we still have to hash them to map the IDs to valid indices**.  
  
\* **Our Hash Table doesn’t handle collisions**. So if we have a collision, we won’t add the value, we’re just going to print something out saying that the array position is already taken.  
\* We could have for example had an array of 26 and then we could have looked at the 1st letter of the last name, that would also lead to collisions of course.  
\* In practice it can be difficult to come up with a hashing function that’s never going to give you collisions.  
=> **Fortunately, we have strategies for dealing with them**.

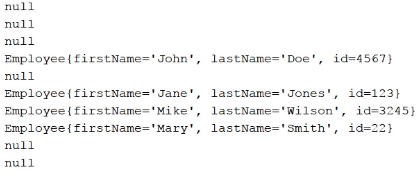
  
**O(1)** => You can see we’re just doing good old array direct access.

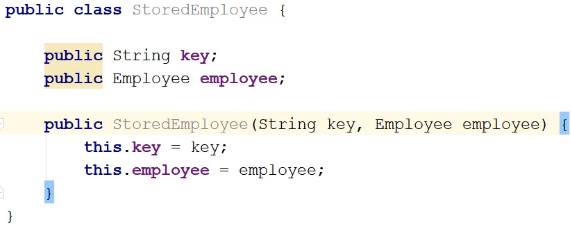
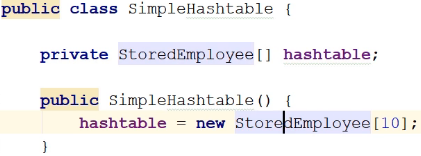
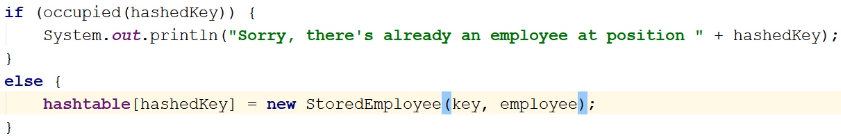
  
\* In our implementation ther will be many collisions and we’re not handling them, so this is sort of a bare bones implementation of a Hashtable to give you an idea of how we do the hashing under the covers.  
\* **The beauty of HashTables is you can use non-integer keys to retrieve items in O(1) constant time**.  
\* The **Load** **Factor** and the **Hashing** **Function** are going to influence the performance.  
\* Now let’s take a look at strategies that are used to handle collisions.

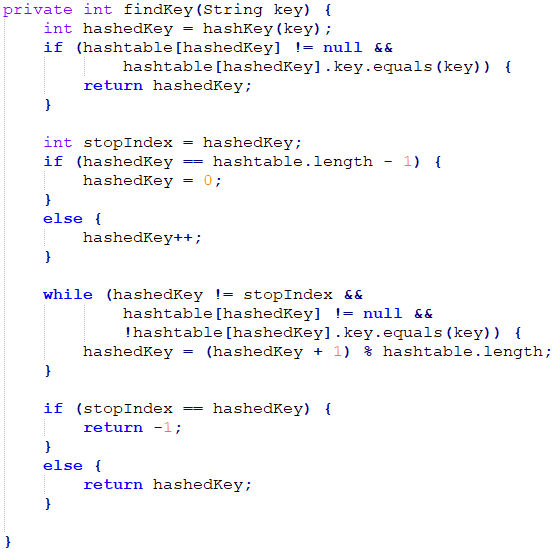
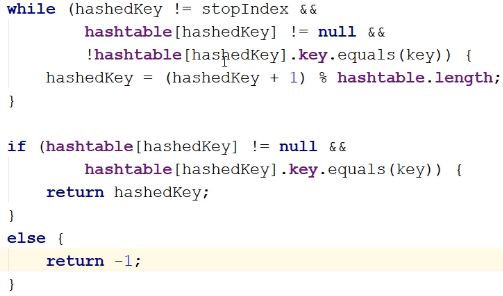
**Linear Probing**  
\* We’re going to take a look at 2 common strategies for handling collisions.  
1) => **Open Addressing**  
=> If we try to put an employee into the table and we find out that there’s already an employee at the slot where we want to put the new employee, then we look for another position in the array.  
=> There are different ways that we can look for other positions.  
=> We’re going to use **Linear Probing**.  
=> When we discover that a position for a hashedKey value is already occupied,   
we **increment that hashedKey value by 1** and then we check the resulting index and if it’s also occupied, we **keep doing that until we find an empty slot or we’ve checked the entire Hash Table**.  
\* It’s called Linear Probing because each time we increment the index, we’re doing it in a Linear fashion and **every increment of the index** is called a **Probe**.  
\* If we had to increment the index 3 times, then we had to use 3 Probes.  
=> **The lower the number of probes, the better**.

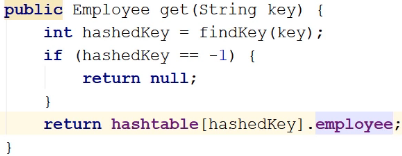
  
  
=> **This implementation is going to wrap if we hit the end of the array and search from 0**.  
=> But we need to know when to stop because then we’re going to be traversing the array from 0 and we don’t want to traverse the part that we’ve already checked.  
=> We can use the hashedKey as the stop index because we’ve already checked that.  
\* index 9 + 1 = 10 % 10 = 0 => this wraps the array, the first IF/ELSE only set the initial value.

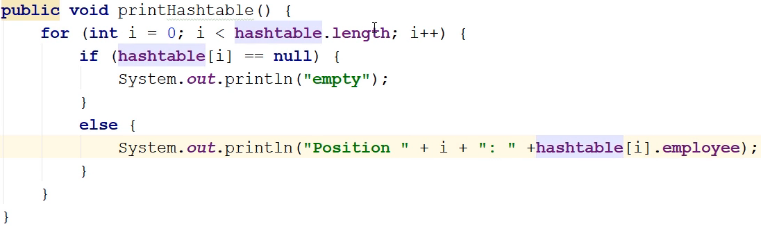
=> When we drop out of the Probing, either we found an empty position or we came out of it because we’ve looked at every position and so if the position is occupied, that means the array is full, otherwise we go ahead and add the employee.

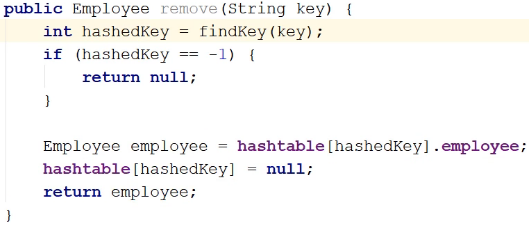
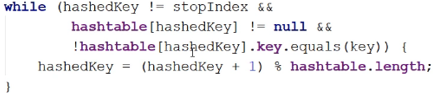
\* If after coming out of the Linear Probing the position is still occupied, then there’s no empty slot in the array. It’s a redundant check if Linear Probing doesn’t happen but it’s in constant time so it’s not all that big of a deal.  


\* We need to update the get() method now.  
=> We have to look and see: was the employee at that position added with the key that is being used to retrieve an employee?  
=> To find it, **we need to store both the key and value in the array**. Because otherwise we have no idea what keys the employees were added with.  
\* Let’s create a StoredEmployee class and make the fields public so that I don’t have to create sets and gets because that’ll mean that the code in the Hashtable won’t be so cluttered.  
  
\* Now we need to update our Hashtable to use the StoredEmployee.  
  
\* And in the put():  


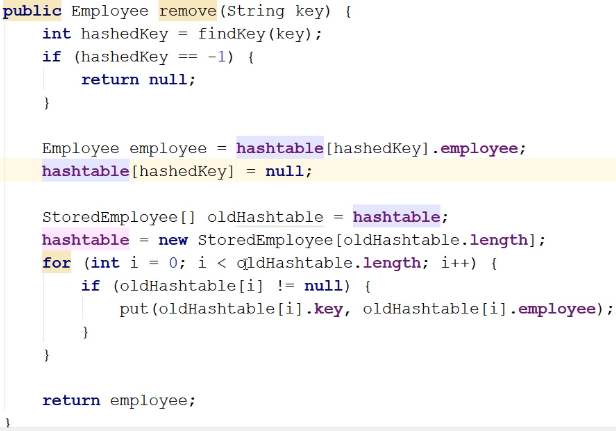
\* Now let’s create a method to find the hashedKey that we’re looking for if it exists.  
  
**!!!!! (there’s a bug here after the WHILE, we’re going to fix it):** **^ fixed**  
=> If the key that was used to store the employee at hashedKey is equal to the key that we’re using to retrieve the employee, then we can just go ahead and return the hashed value because that means that the employee at hashtable[hashedKey] was stored using the key that we’re looking for.  
=> If that’s not the case, we have to use Linear Probing.  
\* Now let’s update the actual get() method:

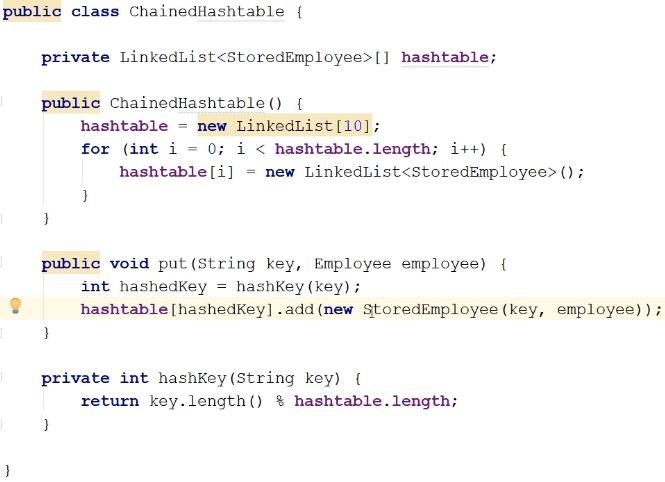
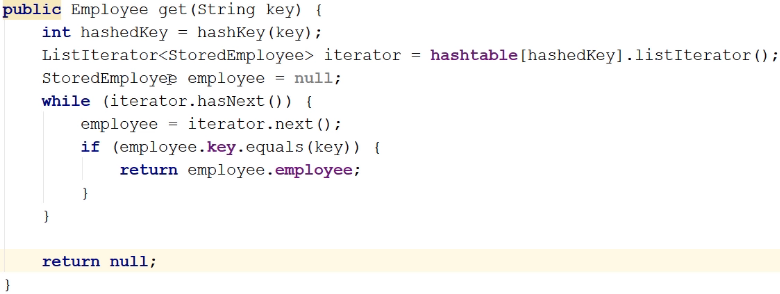


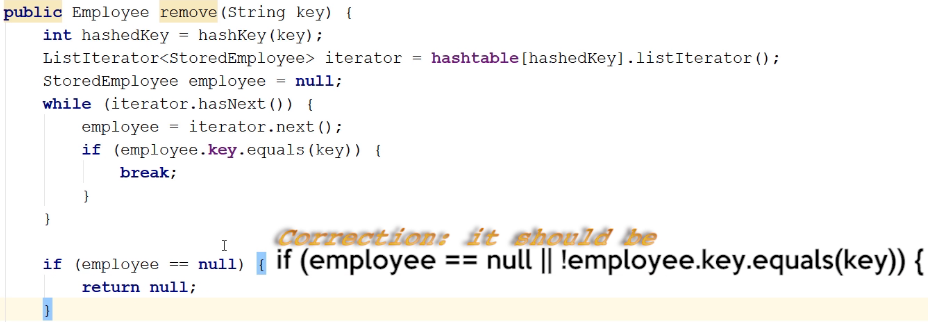
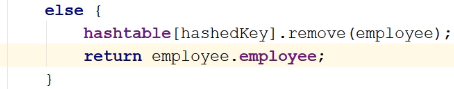
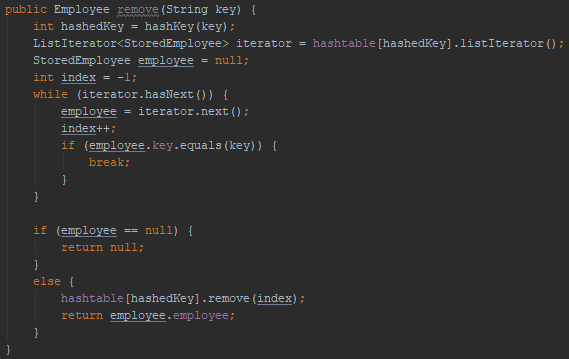
\* That’s it.  
\* Let’s also update the print method.  
  
=> Now we have to use .employee so we need to check for null.  
\* That’s it, that’s Linear Probing.  
\* As you can see, **handling collisions complicates things and it also affects performance**.  
=> **If we have to use probing** - **if there is a collision**, then adding an item into the Hash Table is no longer O(1), the **worst case** is **O(n) because we might have to search the entire table**.  
\* But as I said, in practice it can be difficult to come up with a hashing function that’s going to return unique values and a lot of times the amount of space you have in the data structure that’s backing the table, might not be large enough to hold all the items you want to add, in which case it doesn’t matter how good your hashing function is, collisions will be unavoidable.  
\* We talked about the **Load Factor - that can obviously also affect how many collisions** you’re going to have and **how well Linear Probing works** because if your array is almost full, then you’re going to have to be using a lot of Probes to find an empty spot.  
\* But if you **keep the Load Factor reasonable**, and so you’ll **occasionally resize the array** to create more empty spots, then that will cut down on the average number of probes that you have to use.

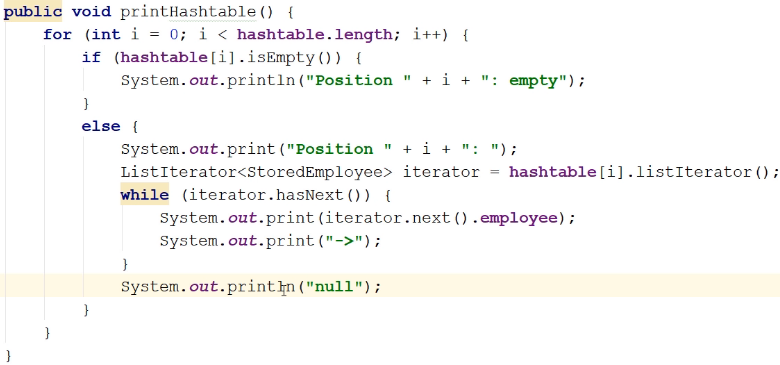
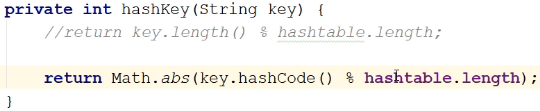
**Linear Probing - Removing items**  
  
\* **In our implementation is not complete, we’re not fully handling removes - for example Jones, Smith - if we search for Smith, we’re going to get the index of Jones and then use Linear Probing to get to Smith. But if we remove Jones, then that index will point to null and we won’t get to Smith**.  
\* You might be asking why do we drop out as soon as we hit null:  
  
=> The reason that we’re doing that is when **we use Linear Probing**, which means that we’re incrementing the index we’re checking by 1, we **always end up putting the item in the first available position after the hashedKey value**.   
=> Let’s say out hashedKey value is 5 and we find out that 5 is full, so we increment 5 to 6 and if we find out 6 is full, then we increment to 7, it’s full, we increment to 8 and we find out it’s empty so we put it there - so when we use Linear Probing we always end up putting the item in the first available position and that’s even true if we have to wrap around to the front.  
\* And so when it comes to getting the item back where we use Linear Probing again, if we hit a null value before we have found the item we’re looking for, we can be certain that the item is not in the table.  
\* **In the next video, we’re going to discuss 2 ways that we can deal with this problem**.  
\* So once we’ve completed this implementation using one of those 2 ways, we’ll never be in a situation where we can have a null value inbetween the hashedValue of the key and the index that we located by using Linear Probing.  
\* And so that’s why it’s okay in this loop to drop out as soon as we hit null. Because we started at the hashedKey and then our first Probe value will either be 0 if we’re wrapping around right away, or 1 greater than the hashedKey and if that position is null, then we know that there isn’t an employee in the table with the key that we’re looking for. Because if there was, there would be no null values between the original hashed position and the position where we inserted the employee.  
\* **So we have to do more in our remove() method**.

**Linear Probing - Rehashing**  
\* There are 2 ways to solve the null inbetween problem:  
1) => **rehash** **all of the items that are already in the Hash Table**.  
\* So you’d create a new hashtable array and then you’re going to loop through the existing array and you’re going to rehash all the values. Because after we’ve deleted Jones, that would move Smith into position 5 because when we do the rehashing, when we come to Smith, we’re going to hash her key which is going to end up being 5, and in the new hashtable, 5 would be empty.  
\* That means that on every remove, you’re going to have to iterate over all of the remaining elements and rehash them and there’s a performance hit and removes will no longer be done in constant time. Even if the employee you’re removing is at the position you expect them to be at, we’re still going to have to iterate over the entire array and rehash everything.  
\* So it has its disadvantages.  
2) => instead of nulling out the position in the array, we could add a field to the StoredEmployee class that says whether an employee has been deleted or not.  
=> This would let us distinguish between positions that have always been empty, meaning nothing has ever been added to them and positions that used to have something in them but that something was deleted.  
=> So when deleting an employee, we wouldn’t actually delete it, we would just set its deleted field to true.  
\* **Of course this has disadvantages**:  
=> you’re going to end up with a polluted Hash Table because you’re going to have a mix of live and deleted values.  
=> the other thing is it means that even though you’re deleting items, your Load Factor isn’t going to change and it means you’re going to have to be doing a lot more Linear Probing because when you go to add an item, if it collides with a deleted item, then you have to do Linear Probing.  
\* In the **first solution**, you’re going to be **hit when you do the remove()** because you’re going to have to rehash the entire Hash Table.  
\* With the **second solution**, you’re going to be hit when you do add() because there’s going to be more collisions and you’re going to have to do a lot more Linear Probing because you never remove anything from the table essentially. **So it hurts both get() and put()**. **Higher Load Factor and collisions**.  
=> So it depends on where you’re willing to take the hit.  
=> I would rather take it on the remove(), I would rehash the table.  
\* **rehashing** => **That will keep the table clean and it means that it will reduce the number of collisions and the Load Factor will be helped by this as well**.

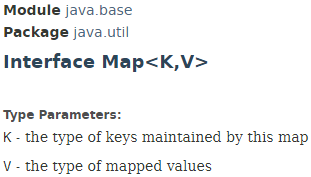
  
\* We could of course have code for resizing the array and a robust Hash Table implementation would have that, in the put() we’d be checking the Load Factor and if necessary, resizing the array and when we do the resize, we’d rehash everything.   
\* For the 2nd solution, that would mean that because we’re keeping a lot of deleting items around, we could potentially be using a lot more memory because we never get rid of anything.  
\* I like the rehashing better for this simple implementation.  
\* That’s it.  
\* And so as you can see, when you use Linear Probing, there’s a lot more work involved. In the best case using Linear Probing is still constant time O(1) if the position you want to work with is empty and so if you’re adding an employee - the position is empty, and if you’re getting an employee - if the key at the hashed position is the one that you’re interested in, that’s constant time.  
\* If you have to use Probes, then the worst case is linear time O(n) because it’s possible that you might have to traverse the entire array before you either find an empty position in the case of an add() or before you find the key you’re interested in.  
=> Does this mean that Linear Probing isn’t good to use to deal with collisions? Well no. Because with a decent hashing function and if you are resizing the array occasionally so you’re keeping the Load Factor reasonable, you’re never going to have to use that many Probes, in the average case you won’t have to use a tons of Probes, you’re not going to have to traverse the entire array.  
\* But as I said, **the hashing function and keeping the Load Factor reasonable is what’s going to make or break your Hash Table’s performance**.  
\* So if you have a hashing function that produces tons of collisions all the time, or your Hash Table is really full and you don’t resize it very often, your performance is going to suffer.  
**Quadratic Probing** => a variation of Linear Probing, instead of incrementing the hashedValue by 1, you increment it by some constant squared. So for example you start out by incrementing the hashedValue by 1 squared and then you’d increment it by 2 squared, and then 3 squared etc.

**Chaining**  
\* Chaining is **a lot simpler than Linear Probing**.  
\* When we use chaining, instead of storing the value directly into the array, **each array position contains a** **Linked List**.  
\* So instead of storing the employee instances, we store a Linked List.  
\* And if we go to add() an employee and the key that we use has a hashedValue that collides with the hashedValue for another key, well, that’s okay because at that position in the array there’s a Linked List and Linked Lists don’t have any boundaries, they’re not bounded by size.  
\* And so we don’t havea to worry about incrementing indices and all that stuff.  
\* The **drawback** is there’s a Linked List at every position.  
=> And so when you go to retrieve or delete an item, **you have to search the Linked List** to find the item with the key you’re interested in.  
=> But if you have **a good hashing function and a good Load Factor**, then these **Linked Lists will typically be short**.  
  
\* **We could add the new LinkedLists in the add() method but doing it up front just simplifies things**.  
\* The other way we could potentially implement this, and you might see this if you look at other implementations, is that we **create an Object array** and then **we store either a StoredEmployee** if there’s only 1 item or **once we get a collision, we then change that to a LinkedList**.  
\* It doesn’t matter here whether we add employees to the beginning or the end of the list because we have no idea in what order they’re going to be retrieved.  
\* We don’t have to worry about the position already being occupied.  
\* So it’s much much simpler but it does **require more memory** because you have the **next/previous** fields.  
  
**ListIterator<> listIterator()  
ListIterator hasNext()  
ListIterator next()**

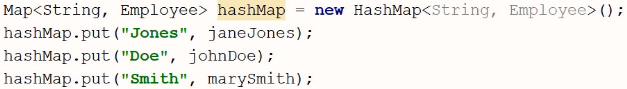
  
  
\* When we break out of the loop, there are 2 possibilities: we have completely traversed the list and we haven’t found a StoredEmployee instance that matches or the StoredEmployee employee field now contains the employee we’re interested in.  
\* **We could of course pull the ListIterator part so that it’s in a separate method**.  
\* **But we can improve this remove() method**.  
=> The way we’re doing it now we’re actually going to end up iterating over the list twice because we’re iterating over it in the WHILE loop to find the employee that has the key and then when we call the remove() method from the LinkedList, the remove() method in the LinkedList class is going to have to iterate over the list again to find the employee object. Because remember in the JDK LinkedList we’ve only got the head and the tail of the list and so it has to traverse the list. And we already traversed it in the WHILE loop.  
=> So we’re going to change this code to keep track of where we are in the list and then there’s another form of the remove() method where we can pass an index. And depending on how the list is implemented in the backend, that can be faster. It may still have to traverse the list but at least it’s not going to have to do all the comparisons again because with the regular remove(), it has to traverse the list and for every list element, it has to compare it.  
  
\* That’s it.  
\* Using LinkedList makes the implementation easier because there’s always room at the hashed location.

\* But that doesn’t mean that Chaining is better than Linear Probing - in fact,   
**on average, Linear Probing performs better than Chaining does**.  
\* It just means that the implementation is a little bit simpler.  
  
**O(n)** => Time Complexity of retrieving an item based on a key.  
=> In the worst case, all the instances would have the same hashedValue and so in that case you would have to potentially traverse every single item in the Hash Table.  
\* But the **important thing here** is the **Hashing algorithm** - we want a hashing function that will result in the smallest number of collisions.  
\* Now of course the best case is no collisions at all but that can be difficult to achieve in practice.  
\* But in the average case, with a decent hashing function, your Linked Lists aren’t going to be that long and so if we say that K is the average length of a LinkedList in your hashtable, then retrieval will be **O(1 + k)**.  
=> The 1 corresponds to calculating the hashedValue and going to that array position  
=> The K provides us with an idea of the average length of a Linked List.  
\* **So whether the Time Complexity is going to be closer to O(1) or O(n) will depend on the hashing function and also on the Load Factor**.  
\* Why does it depend on the Load Factor in this case?  
=> If you’re trying to squeeze 100 values into an array of 10, you’re going to have a much fuller array in terms of Linked Lists that are actually having values than if you try to squeeze 100 items into an array of size 50 because remember in the hashing function we’re always dividing by the hashtable.length in our case, so if this gets bigger, your values will be more uniformly distributed.  
\* For example when our array is of length 10, the numbers 5 and 15 are both going to go into position 5. But if we doubled that array to length 20, then 5 would go into position 5 and 15 would go into position 15.  
\* And so it’s important to understand the **hashing function** and the **Load Factor** are going to be **important in determining how well your Hash Table performs**.  
\* Now after looking at Chaining and Linear Probing, we can see why having a hashing function that doesn’t produce lots of collisions will result in better performance.  
\* One final note - in our hashing function we’re using key.length().  
=> **We could be calling the hashCode() method in the String class**.  
  
=> This is often how you’ll do it. You’ll actually call the hashCode() method in the class.  
**Math.abs()  
String hashCode()**  
\* The hashCode() method for String class can produce negative numbers, so we use Math.abs().

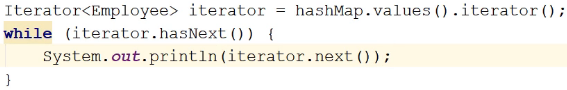
\* **In a hashing function, you’re always going to be modding it by the size of the array because you have to make sure that the values you’re returning are valid array indices**.

**Hashtables and the JDK**  
**Map interface** => The primary interface for Hash Tables in the JDK  
<https://docs.oracle.com/javase/9/docs/api/java/util/Map.html>  


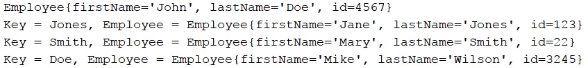
\* It’s an object that maps keys to values.  
\* Map cannot contain duplicat keys, each key can map to at most 1 value.  
=> It doesn’t mean that there cannot be collisions - it’s not talking about the hashed values, it’s talking about   
=> In our case if we were to add an employee with the key “Smith” and then another employee with the key “Smith”, the first employee that we added would be replaced by the second employee.  
=> And so there’s only ever 1 employee in the table that can be associated with the key “Smith”.  
\* And that makes sense because otherwise if we come in and we provide the key “Smith”, there would be no way of knowing which employee we want.  
\* We didn’t handle that in our implementations of Hash Table but our implementations are not robust implementations. If you were going to use those Hash Tables in a real-world application, then we would have a lot more work to do. We’re not handling resizing either.  
**HashMap** => **implements Map** =>   
<https://docs.oracle.com/javase/9/docs/api/java/util/HashMap.html>  
\* This implementation permits null values and the null key.  
\* Provides **O(1)** for the basic operations but **of course if it has to resize the array and do re-hashing, then you’re not going to get constant time**, you get O(n).  
\* There’s a discussions in the docs about the **Initial Capacity** and the **Load Factor**,   
**you can set the Load Factor that you want and when it’s exceeded, the Hash Table is resized**.  
\* The default is 0.75.  
\* **HashMap is not synchronized**.  
=> They suggest you wrap it with **Collections.synchronizedMap()** method if you want to use it with multiple Threads.

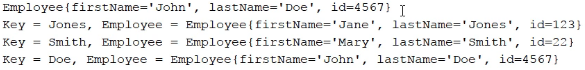


\* Because of course you most likely would not implement your own Hash Table in Java, you would use one of the classes in the JDK.  
\* We can print it using for example:  
**1) => Iterator**  
**HashMap** **values() => gives us the values in the HashMap  
iterator()**





=> It’s important to note that they’re not in the order we added them because where they’re placed in the Hash Table is going to depend on the hashed value of the keys.  
**2) => forEach + lambda expression**  
\* Only available in Java 8+  
**HashMap forEach()**  
  
**HashMap containsKey()  
HashMap containsValue()**  
\* We expect the **containsKey() method to be faster** because we’re passing the key, and so under the convers, the key will be hashed and the method will be able to jump right to that place in the underlying implementation.  
=> **So it’s always better to have the key**.  
  
  
=> **Mike Wilson will replace John Doe**.  
**HashMap** **put() => method returns the employee that used to be there**:  
  
=> **This will return John Doe**.  
  
\* You might not want that to happen.  
\* If you use a key ttat’s already been used, and you don’t want the old value to be replaced, you can use the:  
**HashMap** **putIfAbsent()** **=> will only insert the value if the key hasn’t been used yet.**  
=> This method returns the value that’s in there if the key has been used.  


\* Then we get:  


**HashMap** **get() => if it doesn’t find the item, it returns null.**  
\* You might not want to get null, and so you can use:  
**HashMap** **getOrDefault()** **=> lets you** **specify a default value**.  
  
**HashMap** **remove() => returns the item that is removed.**

\* There are more methods in the HashMap class but I just wanted to give you a taste of what you can do.

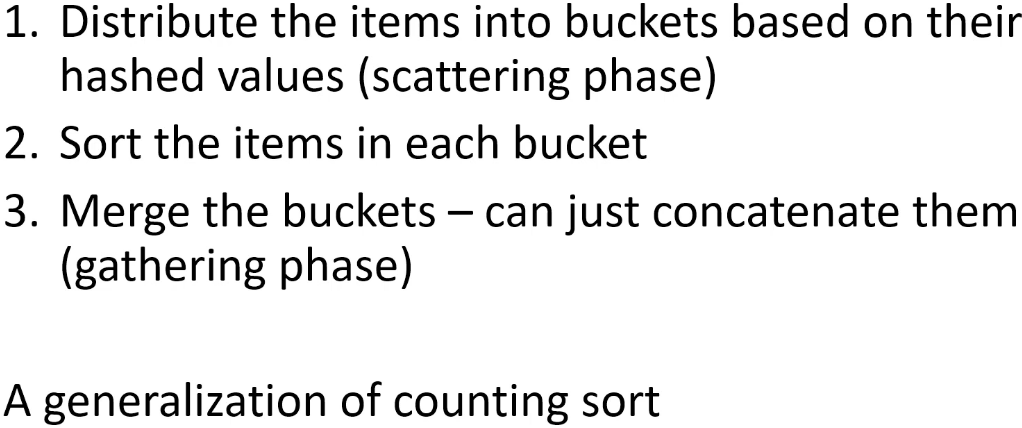
**LinkedHashMap** => **A subclass of HashMap**.  
<https://docs.oracle.com/javase/9/docs/api/java/util/LinkedHashMap.html>  
\* This is a Hash Table and Linked List implementation of the Map interface.  
=> This doesn’t mean that this implementation is backed by a Linked List because it says it’s a Hash Table and a Linked List and also we know that it’s a subclass of HashMap. And HashMap is an array implementation.  
\* They say:  
=> This implementation differs from HashMap in that it **maintains a Doubly Linked List running through all of its entries**.  
=> This Linked List defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order).  
\* So this means that it’s still being backed by an array, but they also have put all of the entries into a Linked List.  
\* The implementation is **not synchronized**.  
\* If you need it, you can again do:  
  
\* Most of the methods will be the same because this implements the Map interface.  
**LinkedHashMap** **removeEldestEntry() => you can specify that you want the map to delete its oldest entry every time you add a new one.**  
\* **Why would you want to do this?**  
=> Well it’s possible that you’re using the Map instance as a **CACHE** and in that case you wouldn’t want the map to just keep growing and growing, because for a cache, (basically cache is for faster access of things that have been used recently), and so that’s why you’d want to remove the eldest or the oldest entry, the one that’s been in the list the longest.  
\* So if you decide that you’re going to use this class to implement a cache and you only ever want to keep let’s say 100 items, then if it’s full and you want to add another item, you’re going to take out the oldest item and add the new item.  
\* **The cache is there to provide quick access to recently used items**.  
\* The oldest item is going to be the most stale item and so it’s the one you want to remove.

**Hashtable**  
<https://docs.oracle.com/javase/9/docs/api/java/util/Hashtable.html>  
\* This differs from a HashMap in a couple of ways:  
=> **You can’t add null key or values**.  
=> **Synchronized.**

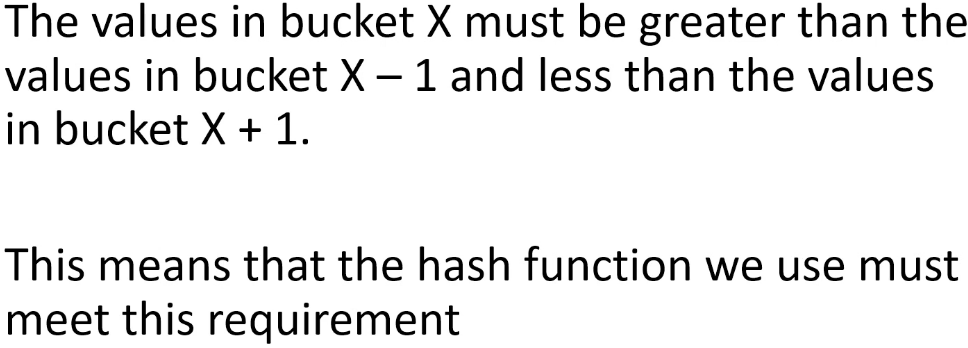
**ConcurrentHashMap**  
<https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ConcurrentHashMap.html>  
=> **Synchronized**.  
=> **Supports full concurrency of retrievals and high expected concurrency for updates**.  
\* And so **if you really wanted concurrency, and you were going to deal with a lot of items**, this would be a good class to use.

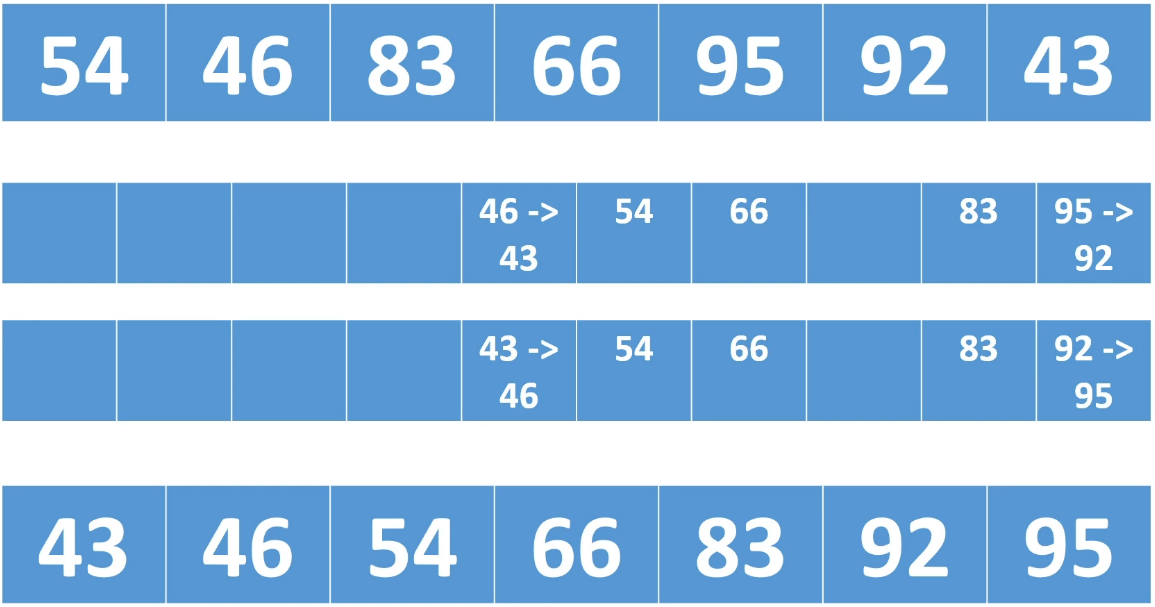
\* There’s a lot of classes that implement the Map interface, they’re usually used under specific circumstances, the ones that we haven’t looked at.  
\* There’s a lot of support for Hash Tables in the JDK so if you’re working on an application and you’re thinking you need a Hash Table, then take a look at what your requirements are and take a look at the classes that implement the Map interface to see which one is going to best suit your needs.

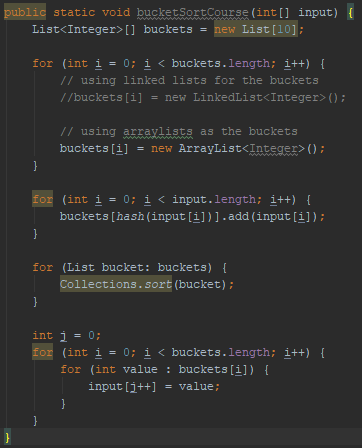
**Bucket Sort (Theory)**  
**Bucket Sort** => Now that we know about Hash Tables and more important we understand what Hashing is, we can look at another Sort Algorithm.  
**Not in-place => it uses extra memory that depends on the number of items in the array that you’re sorting.**  
**Stability** **will depend on sort algorithm used to sort the buckets** - ideally, you want a **stable** **sort**.  
**=> To achieve O(n), must have only 1 item per bucket**=> **Insertion Sort is often used to sort the buckets because it is fast when the number of items is small**.  
=> **Uses hashing**  
=> **Makes assumptions about the data, like Radix and Counting sort**=> **Because it makes assumptions, can sort in O(n) time**  
=> **Performs best when hashed values of items being sorted are evenly distributed, so there aren’t many collisions**  
\* When it comes to Bucket Sort, we’re hashing the values that we are sorting.  
\* So there is no concept of keys and values.  
\* The important thing is the values that we are sorting are hashed.

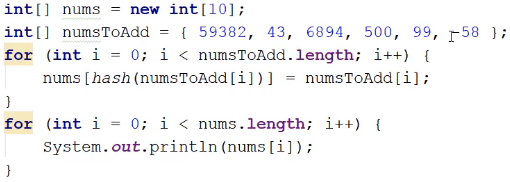


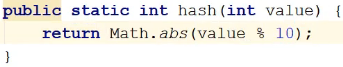
**Scattering Phase**  
**Gathering Phase**  
\* Because all of the items in every bucket have been sorted, you can just basically concatenate all the buckets one after the other.  
\* **And so in the 3rd phase you’re going to be copying the items in the buckets back into the original array**.  
\* This is actually a generalization of Counting Sort, because if you remember Counting Sort, we went through the values and then we distributed the values into the Counting Array and then in the final phase we just traversed the Counting Array and for each value we copied how many we had, back into the original arrray.  
\* Bucket Sort is doing something similar, except it’s distributing the items based on their hashed values and then it has to sort the items in each bucket. But then it goes over the buckets and copies them back into the original array.

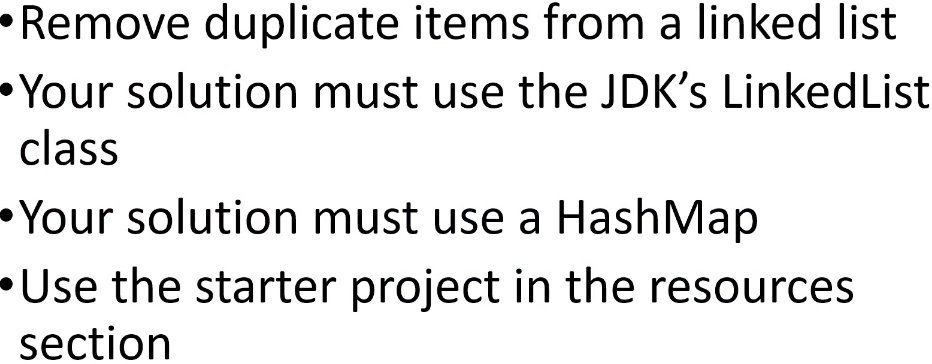


\* **There’s one important thing here** - in order for the **Gathering Phase** to work, the values in bucket X must be greater than the values in bucket X - 1 and less than the values in bucket X + 1.  
=> This means that in the merging phase, we are going to write the values in bucket 0 back to the arrya nd then we’re going to follow those by the values in bucket 1 and then values in bucket 2 and so on.  
=> That means that the values in bucket 0 all have to be < less than the values in bucket 1.  
=> Otherwise when we write the values in bucket 0 and then bucket 1, they’re not going to be sorted.  
=> This means that **whatever hashing function you use, it has to make sure that the hashed values it produces meet that requirement**.  
\* So if you’re sorting values 1 2 3, you can’t have a hashing function that would put 3 into a lower bucket than 1.  
  
**1st array => original array  
2nd array => Scattering Phase  
3rd array => Sorting Phase  
4th array => Gathering Phase**\* Let’s say we are going to sort the array at the top.  
=> We’re going to sort them into buckets based on their 10’s position.  
=> And so the way the hash function is going to hash these values is hash   
54 to 5,   
46 to 4,   
83 to 8,  
66 to 6,  
95 to 9,  
92 to 9,  
43 to 4,  
\* And so, in the buckets, 46 and 43 are going to end up in the same bucket.  
\* 54 will end up in the bucket for the 50s.  
\* 66 will end up in the bucket for the 60s.  
\* Etc.  
\* This meets the requirement that all of the values in the lower bucket have to be less than all the values in the higher bucket.  
\* In our implementation, we’re going to use the sort() method in the Collections class in the JDK, it uses an adaptive merge sort algorithm.

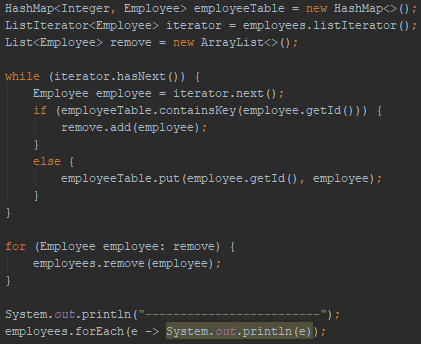
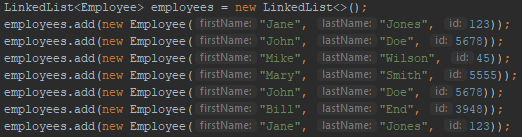
**Bucket Sort (Implementation)**  
**(Implemented my own version before watching)**  
\* For this implementation, each bucket will be an ArrayList.  
\* We’re using an ArrayList, you could have a LinkedList instead.  
  
\* Because we have the buckets as a List, it should be fairly simple to change this to a LinkedList by just changing the ArrayList to the LinkedList.  
\* So when you use the parent interface for the type rather than a specific type, that’s how easy it is to change from one class that implements the interface to another class.

**Hashtables Challenge #1**  
\* A challenge to test your knowledge of what a hash function should be doing.  
\* All I want you to do is write a hash() function.  
=> We have an array of integers and it’s length 10.  
=> I want you to add these integers into that array:  
  
=> And I want you to add those to the array so that you would be able to retrieve these numbers from the array just by being given the number.  
=> To do that, you’re going to have to hash the numbers.  
\* Don’t worry about collision, I’ve deliberately chosen numbers that won’t collide with each other because the purpose of this challenge is just to make sure you understand what a hashing function is supposed to be doing and how to write one when you want to be able to hash values to array indexes.

**Hashtables Challenge #1 Solution**  
**(Implemented the challenge)**  
\* The key here is that the hash function has to return valid array indices.  
\* We know our array is of length 10 and so the valid indices are 0 - 9.  
\* Usually when you’re writing a hashing function and you’re wanting to hash the values to array indices, you want to mod % by the length of the array because that guarantees that the value you’ll get is going to be between 0 and the last valid index.  
  
\* If we wanted to get one of these values if they were keys for example, we would run the same hash function on the values and that would give us the index.  
\* So if we were passed 500, we would call the same hash function, we get 0 back and then we would find it in nums[0].

**Hashtables Challenge #2**  


\* In your solution you shouldn’t have to maintain any extra fields pointing at items in the LinkedList.  
\* So you shouldn’t have to have any fields, trying to track where things are in the LinkedList as you’re removing duplicates.  
\* Instead, your solution should use a HashMap and you need to figure out how to use that HashMap.  
\* One important thing here is that   
**you can assume that if an employee has the same key as another employee, they’re duplicates**, and so you don’t need to check the entire instance, you don’t need to compare the IDs and first names and last names.  
\* So for this challenge if 2 employees have the same ID, they’re the same employee.

**Hashtables Challenge #2 Solution**  
**(implemented the challenge)**  
  
\* I’m going to traverse the list and for each employee, I’m going to check to see whether it’s in a HashMap that consists of key/value pairs, the key is the ID and the value is the employee.  
\* So I’m going to check whether the employee is in the HashMap and I’m going to do that by checking whether it contains the key.  
\* I said that we can assume that if 2 employees have the same key, they’re the same employee. There’s no need to actually compare the instances against each other.  
\* And so all I have to do is check whether the key is in the HashMap and if it is, then I can assume that the employee has been added already and that means it’s a duplicate and so I’m going to want to remove it.  
\* I’m going to use a ListIterator to traverse the list.  
\* And then I need another list and the reason I need another list is as I’m using the ListIterator to traverse the list, I can’t remove a duplicate when I find one because the Runtime will throw an exception - because as I’m using the iterator to traverse the list, the list cannot change otherwise the iterator will throw an exception and say: Hey, I’m iterating over this list, and now you’ve gone ahead and changed it, so I’m going to throw an exception.  
\* And so if I hit a duplicate, instead of removing it right there and then, I’m just going to add it to a list.  
\* And then once I’ve finished traversing the LinkedList, I’ll traverse the list of employees I want to remove and actually remove them from the list.  
\* Your solution doesn’t have to match mine, you might be traversing the list using a different method and so you can just go ahead and remove the employees on the spot, or you might have a little optimization here that instead of adding the employees to the list, you may be tracking their position in the list - if you provide the index that you want to delete, so in at the end we would provide remove() with an index instead of an object, it would be a little bit faster.  
\* As long as you’re using a HashMap and the duplicates have been removed from the LinkedList, your solution is good.  
\* We could use something called a HashSet but we haven’t looked at sets yet. Here because we really just care about the IDs, we could use HashSet, we don’t really need key/value pairs.

**Resources**  
Map interface javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/Map.html>  
HashMap class javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/HashMap.html>  
LinkedHashMap class javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/LinkedHashMap.html>  
Hashtable class javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/Hashtable.html>  
ConcurrentHashMap class javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ConcurrentHashMap.html>