Understanding the importance of the hash code

HashSet and HashMap, are based on the hash codes of objects.

This can be a confusing topic for new programmers, so I want to spend some extra time explaining it.

Since sets are unique because they don't support duplicates, adding an element always incurs the cost of first checking for a match.

If your set is large or very large, this becomes a costly operation, O(n), or linear time, if you remember the Big O notations I covered previously.

A mechanism to reduce this cost, is introduced by something called hashing.

If we created two buckets of elements, and the element could consistently identify which bucket it was stored in, then the lookup could be reduced by half.

If we created four buckets, we could reduce the cost by a quarter.

Understanding the importance of the hash code

A hashed collection will optimally create a limited set of buckets, to provide an even distribution of the objects across the buckets in a full set.

A hash code can be any valid integer, so it could be one of 4.2 billion valid numbers.

If your collection only contains 100,000 elements, you don't want to back it with a storage mechanism of 4 billion possible placeholders.

And you don't want to have to iterate through 100,000 elements one at a time to find a match or a duplicate.

Understanding the importance of the hash code

A hashing mechanism will take an integer hash code, and a capacity declaration which specifies the number of buckets to distribute the objects over.

It then translates the range of hash codes into a range of bucket identifiers.

Hashed implementations use a combination of the hash code and other means, to provide the most efficient bucketing system, to achieve this desired uniform distribution of the objects.

Hashing starts with understanding equality

To understand hashing in Java, I think it helps to first understand the equality of objects.

I've touched on this in previous videos, but now I want to be sure you thoroughly understand this subject, because it matters when dealing with any hashed collections.

There are two methods on java.util.Object, that all objects inherit.

These are equals, and hashCode, and I show the method signatures from Object here.

Testing for equality	The hashcode method
`public boolean equals(Object obj)`	`public int hashCode()`

The equals method on Object

The implementation of equals on Object is shown here.

It simply returns `this == obj`.

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Do you remember what == means for Objects?

Do you remember what == means for objects?

It means two variables have the **same reference to a single object in memory**.

Because both references are pointing to the same object, then this is obviously a good equality test.

Equality of Objects

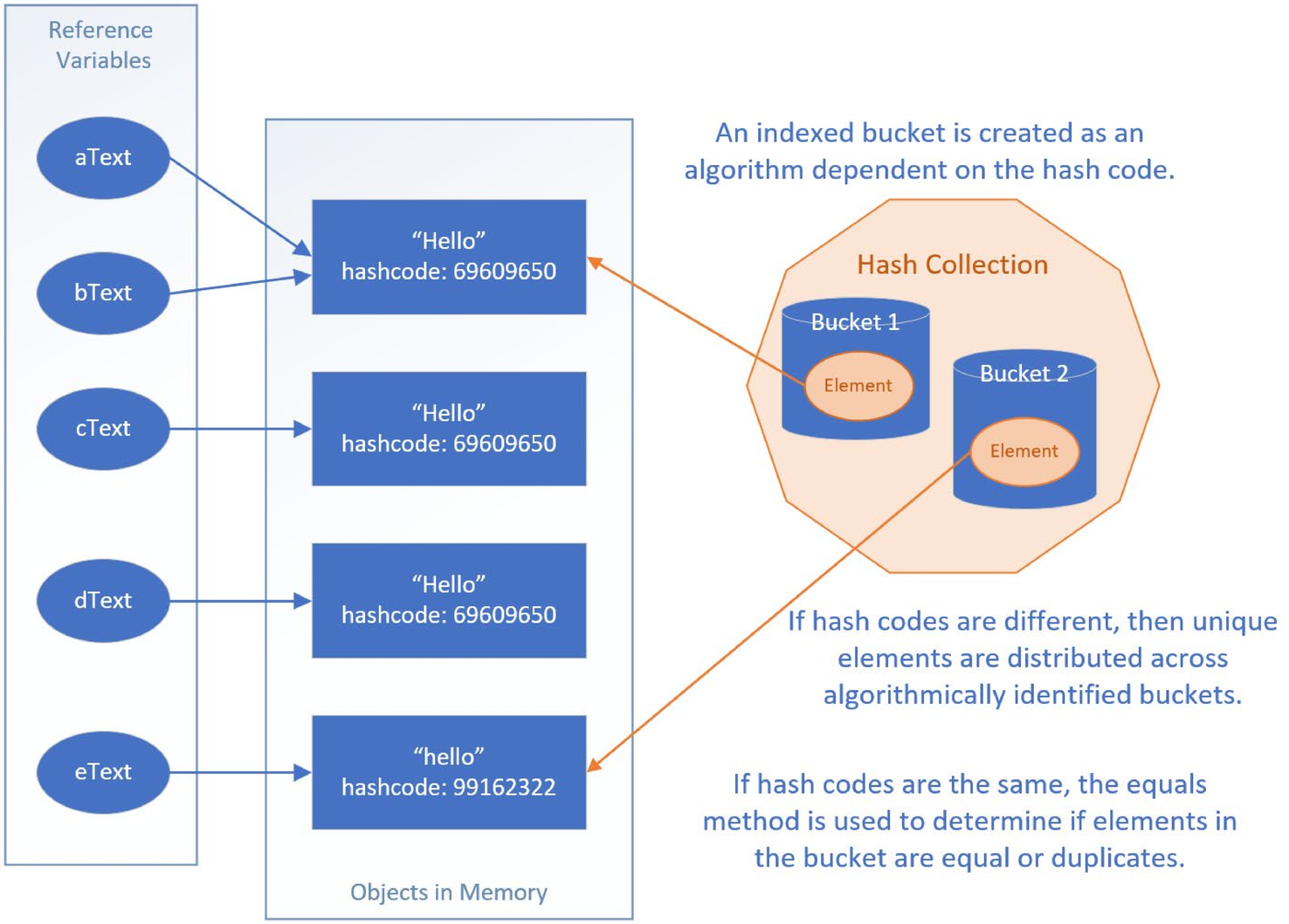
Objects can be considered equal in other instances as well, if their attribute values are equal, for example.

The String class overrides this method, so that it compares all the characters in each String, to confirm that two Strings are equal.

The visual representation of the code

Our code set up five String reference variables, but two of these, referenced the same string object in memory, as shown here with aText and bText pointing to the same string instance.

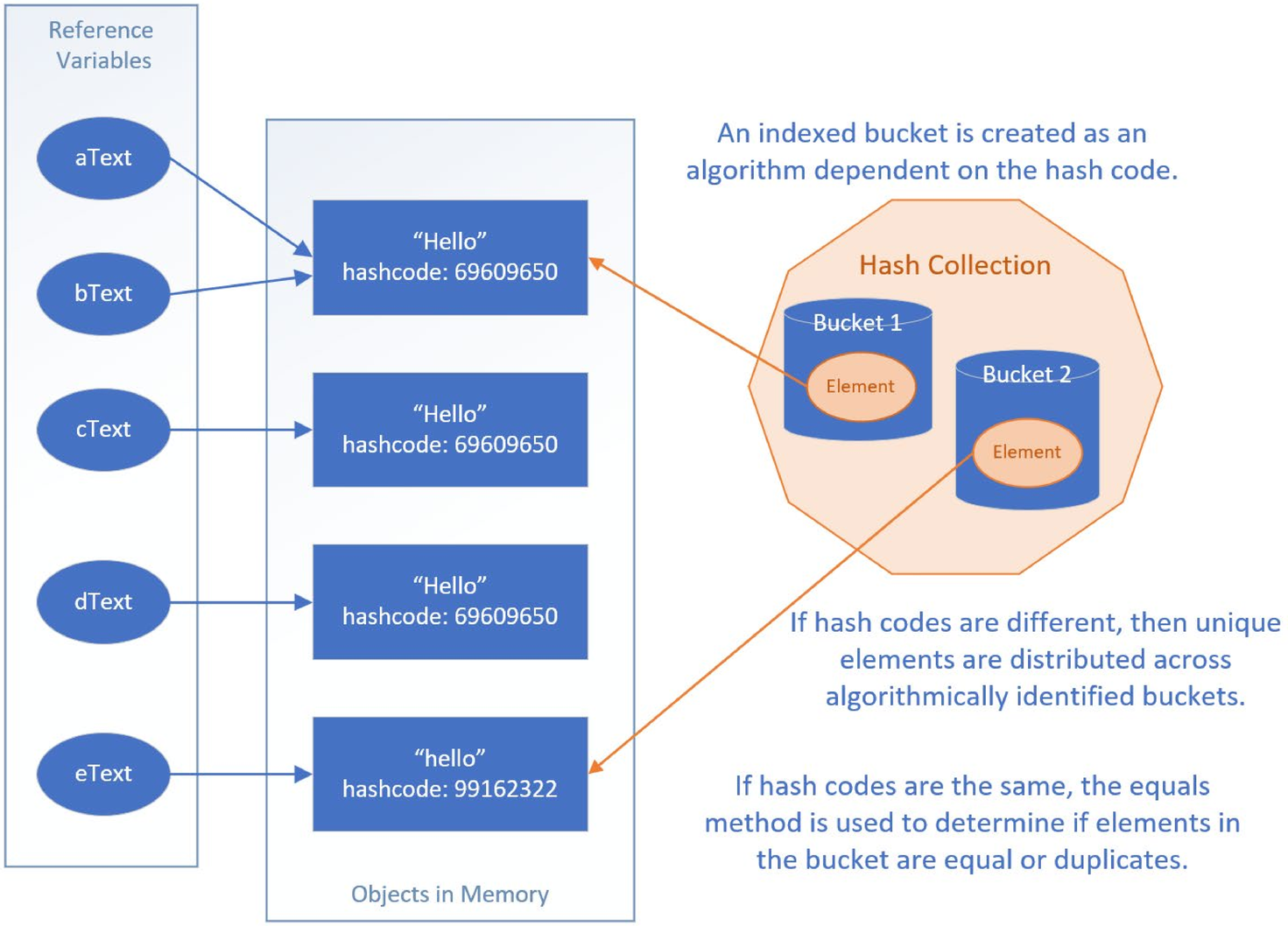
When we passed our list of five strings to the HashSet, it added only unique instances to its collection.



The visual representation of the code

It locates elements to match, by first deriving which bucket to look through, based on the hash code.

It then compares those elements to the next element to be added, with other elements in that bucket, using the equals method.



Creating the hashCode method

You don't have to use the generated algorithm as I did here.

You could create your own, but your code should stick to the following rules.

1. It should be very fast to compute.
2. It should produce a consistent result each time it's called. For example, you wouldn't want to use a random number generator, or a date time based algorithm that would return a different value each time the method is called.
3. Objects that are considered equal should produce the same hashCode.
4. Values used in the calculation should not be mutable.

Creating the hashCode method

It is common practice to include a small prime number as a multiplicative factor (although some non-prime numbers also provide good distributions).

This helps ensure a more even distribution for the bucket identifier algorithm, especially if your data might exhibit clustering in some way.

IntelliJ and other code generation tools use 31, but other good options could be 29, 33 (not prime but shown to have good results), 37, 43 and so on.

You want to avoid the single digit primes, because more numbers will be divisible by those, and may not produce the even distribution that will lend itself to improved performance.

Creating the hashCode method

For those who like to understand how things really work, let me encourage you to do some research on this topic.

For the rest of you, remember that if you are using your own classes in hashed collections, you'll want to override both the equals and the hashCode methods.

I'll be covering this quite a bit over the next couple of videos, so you'll have more time to get exposed to these concepts.

Java's Hashed Collection Types

Java supports four hashed collections implementation, which we'll be looking at coming up.

These are the HashSet, LinkedHashSet, the HashMap, and LinkedHashMap.

In addition there's one legacy implementation, the HashTable, which I won't be covering, since there are more efficient implementations which replace this legacy class.