NPTEL

noc24_cs43

Programming in Java

Live Interaction Session 11: 05th Apr 2024

Understanding the importance of the hash code

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

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, first we need to understand the equality of objects.

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

These are equals, and hashCode, and these method signatures from Object.

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

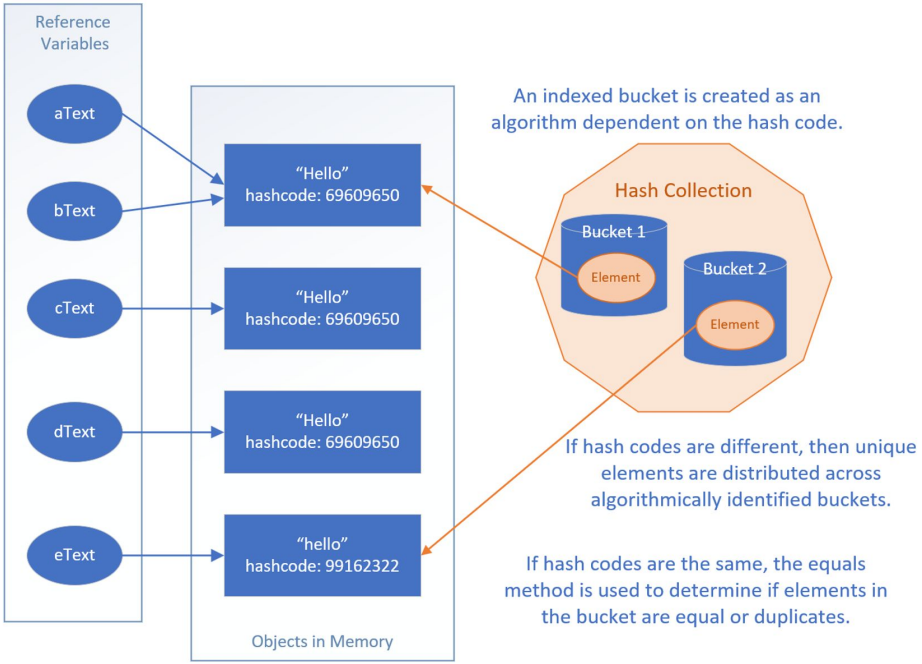
Objects can be considered equal in other instances as well, if their attribute values are equal, for example.

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.

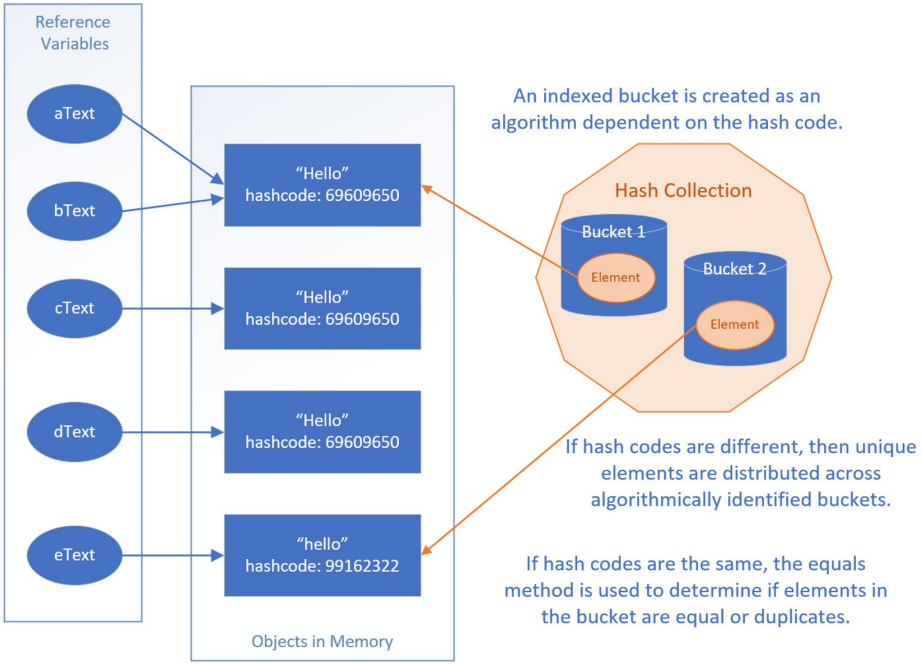
When we passed our list of five strings to the HashSet, it added only unique instances to its collection.



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

- It should be very fast to compute.
- 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.
- Objects that are considered equal should produce the same hashCode.
- 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.

The Set

A Set is not implicitly ordered.

A Set contains no duplicates.

A Set may contain a single null element.

Sets can be useful because operations on them are very fast.

Set Methods

The set interface defines the basic methods **add, remove** and **clear**, to maintain the items in the set.

We can also check if a specific item is in the set using the contains method.

Interestingly enough, there's no way to retrieve an item from a set.

You can check if something exists, using **contains**, and you can iterate over all the elements in the set, but attempting to get the 10th element, for example, from a set isn't possible, with a single method.

The HashSet class

The best-performing implementation of the Set interface is the **HashSet** class.

This class uses hashing mechanisms to store the items.

Oracle describes this class as offering constant time performance for the basic operations (add, remove, contains and size).

Constant time has the Big O Notation O(1).

The HashSet actually uses a HashMap in it's own implementation, as of JDK 8.

Set Operations

When you're trying to understand data in multiple sets, you might want to get the data that's in all the sets, that's in every set, or the data where there's no overlap.

The collection interface's bulk operations (addAll, retainAll, removeAll, and containAll) can be used to perform these set operations.

Representing Sets in a Venn Diagram

Sets are often represented as circles or ovals, with elements inside, on what is called a Venn Diagram.

Here, two sets that have no elements in common.

This venn diagram shows some of the cartoon characters of the Peanuts and Mickey Mouse cartoons.

Because the characters are distinct for each set, the circles representing the sets don't overlap, or intersect.

Representing Sets in a Venn Diagram

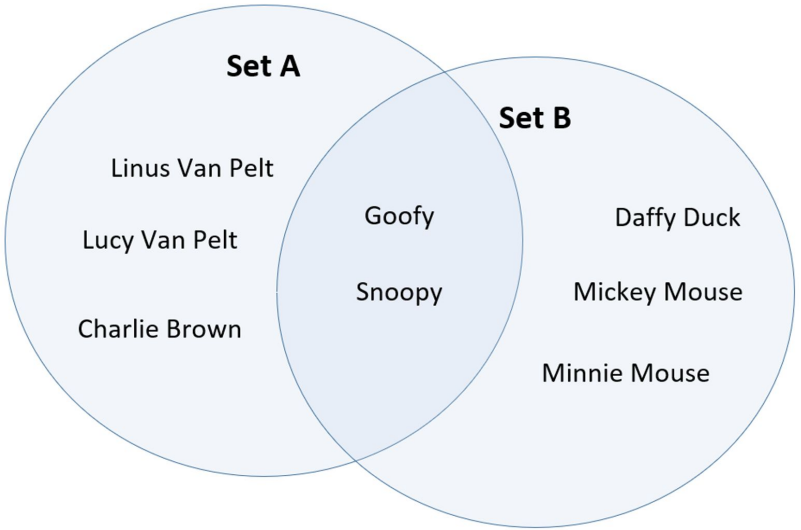
This diagram shows two sets of characters that do overlap.

Let's say that Goofy and Snoopy, have guest appearances in the other's holiday special show.

The intersection of these sets is represented by the area where the two circles (sets) overlap, and contains the elements that are shared by both sets.

Goofy and Snoopy are both in Set A and Set B, in other words.

Venn Diagrams are an easy way to quickly see how elements in multiple sets relate to each other.

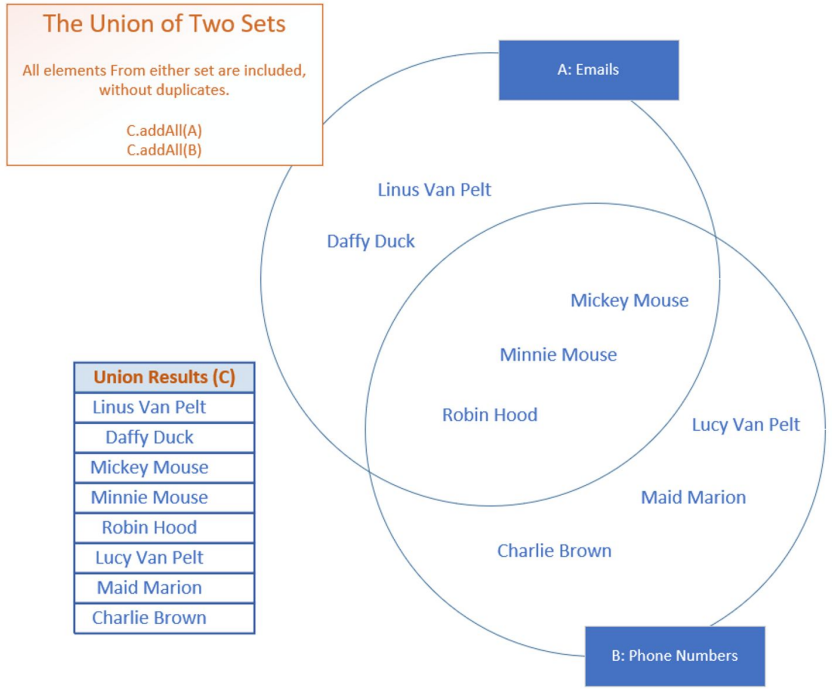


Set Operations - Union A ∪ B

The union of two or more sets will return elements that are in any or all of the sets, removing any duplicates.

The slide shown here is showing two sets, names on an email list, and names on a phone numbers list.

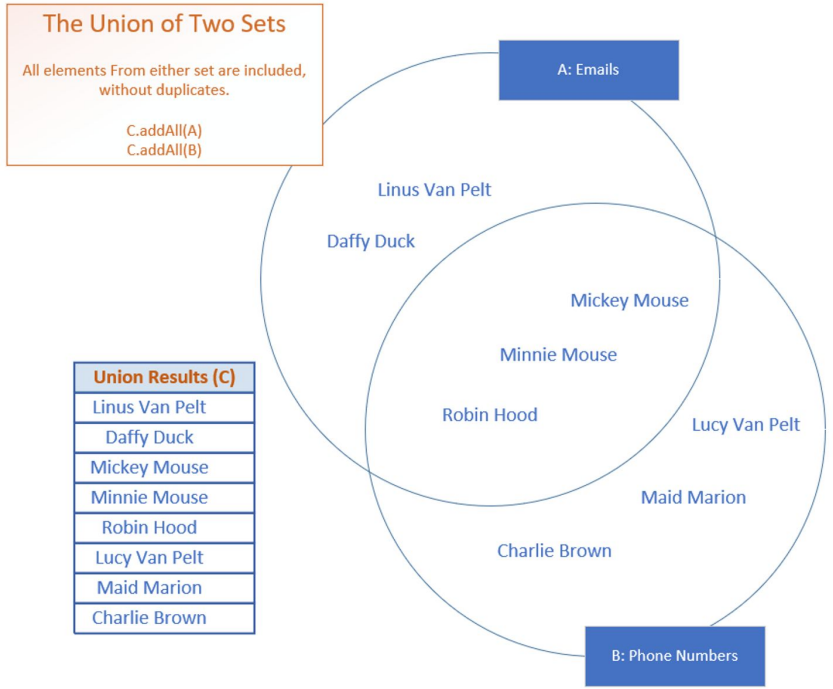
The overlap are names that are on both lists.



Set Operations - Union A ∪ B

In the example shown on this slide, all names on the email list and phone list will be included in a union of the two sets, but Minnie, Mickie and Robin Hood, which are the only elements included in both sets, are included in the resulting set only once.

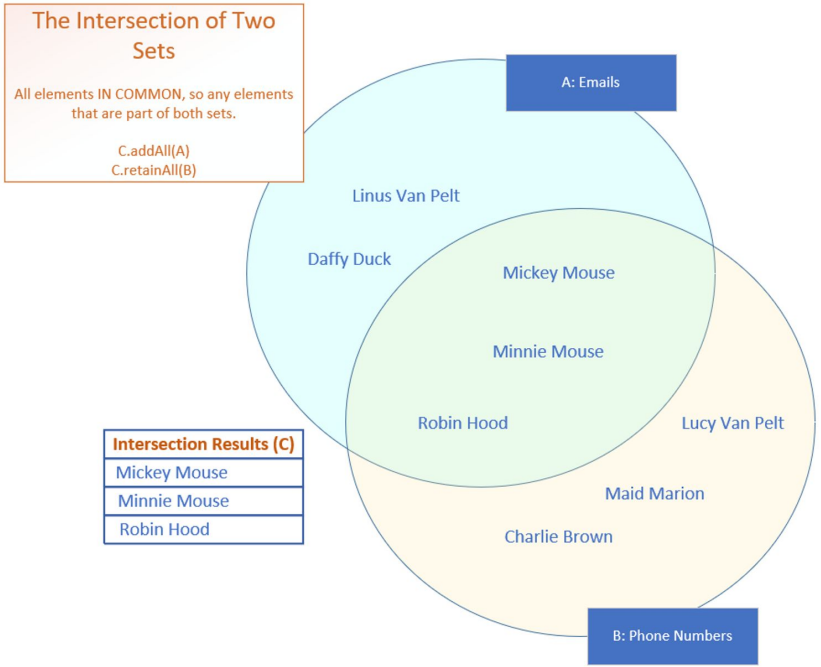
Java doesn't have a union method on Collections, but the addAll bulk function, when used on a Set, can be used to create a union of multiple sets.



Set Operations - Intersect - A ∩ B

The intersection of two or more sets, will return only the elements the sets have in common.

These are shown in the overlapping area of the sets on this slide, the intersect, shown in green, and includes Mickey and Minnie Mouse, and Robin Hood.



Set Operations - Symmetric Operations

The ability to evaluate sets, A intersect B and get the same result as B intersect A, means that the intersect operation is a symmetric set operation.

Union is also a symmetric operation.

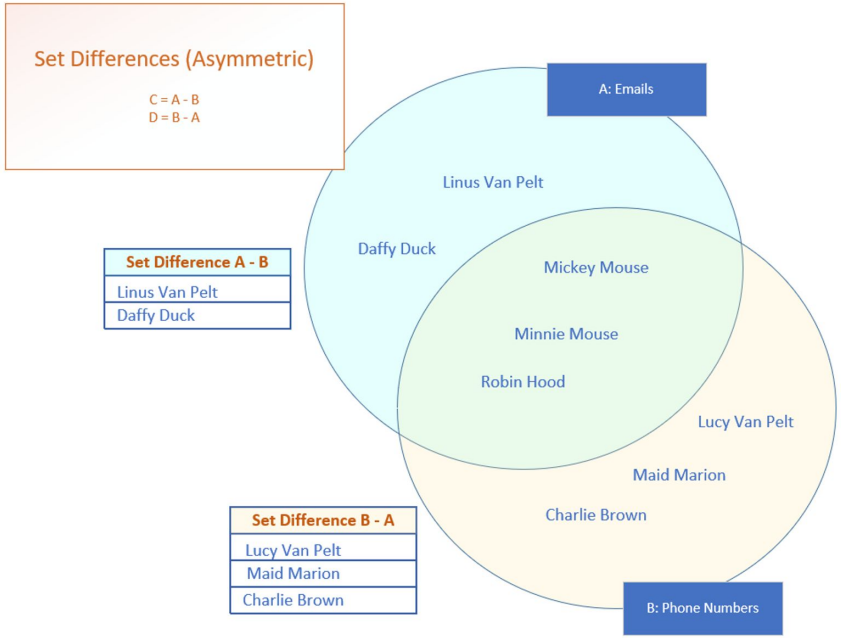
It doesn't matter if you do A Union B, or B union A, the final set of elements will all be the same set of names.

Set Operations - Asymmetric Differences

A difference subtracts elements in common from one set and another, leaving only the distinct elements from the first set as the result.

This is an asymmetric operation because if we take Set A and subtract Set B from it, we'll end up with a different set of elements than if we take Set B and subtract Set A.

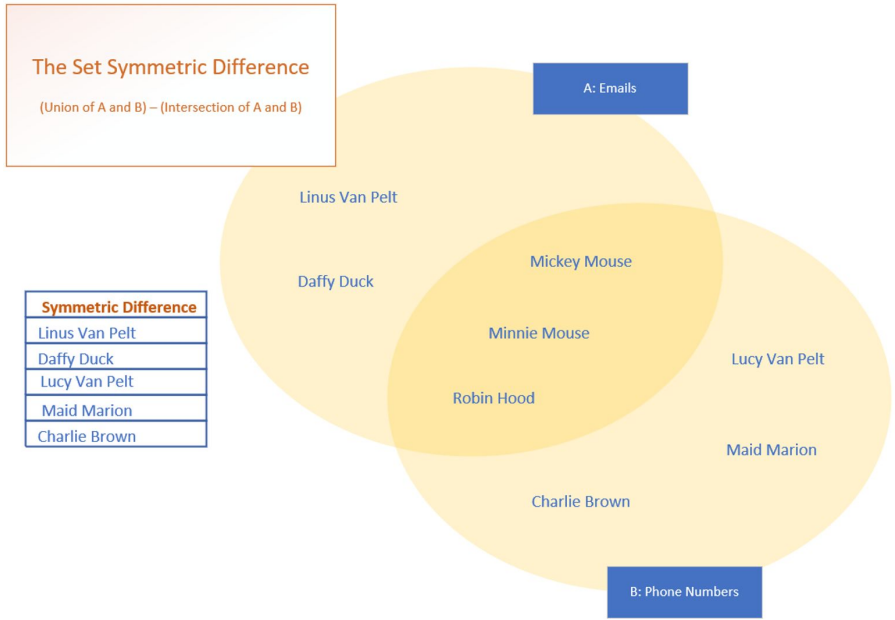
The sets from these two operations won't result in the same elements.



Set Operations - Symmetric Differences

You can think of the set symmetric difference, as the elements from all sets that don't intersect.

On this slide, these are the elements that are represented in the paler yellow areas.



Ordered Sets

If you need an ordered set, you'll want to consider either the **LinkedHashSet** or the **TreeSet**.

A LinkedHashSet maintains the insertion order of the elements.

The TreeSet is a sorted collection, sorted by the natural order of the elements, or by specifying the sort during the creation of the set.

The LinkedHashSet

The LinkedHashSet **extends the HashSet** class.

It maintains relationships between elements with the use of a doubly linked list between entries.

The **iteration order** is therefore the same as the **insertion order** of the elements, meaning the order is **predictable**.

All the methods for the LinkedHashSet are the same as those for the HashSet.

Like HashSet, it provides constant-time performance, O(1), for the add, contains and remove operations.

TreeSet

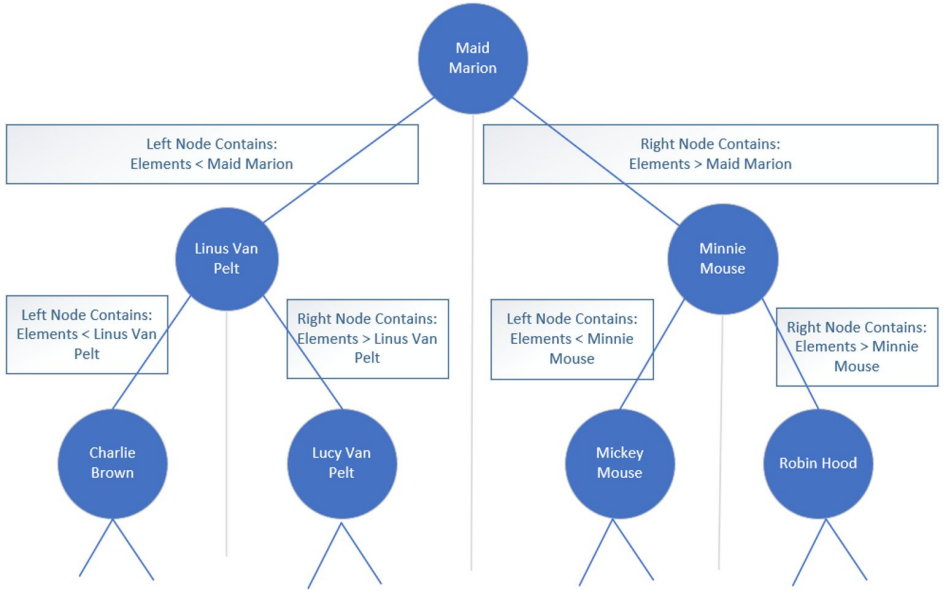
A TreeSet's class, uses a data structure that's a derivative of what's called a binary search tree, or Btree for short, which is based on the concept and efficiencies of the binary search.

The search iteratively tests the mid range of a group of elements to be searched, to quickly find its element, in a collection.

TreeSet

As elements are added to a TreeSet, they're organized in the form of a tree, where the top of the tree represents that mid point of the elements.

This slide shows a conceptual example, using some of the character contacts from my last samples of code.

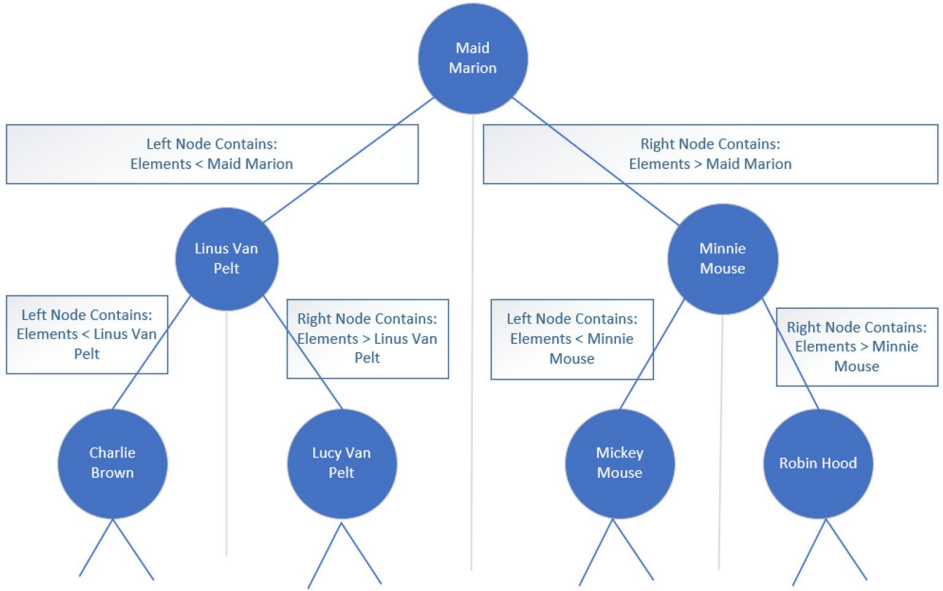


TreeSet

Further binary divisions become nodes under that.

The **left** node and its children are elements that are **less than** the parent node.

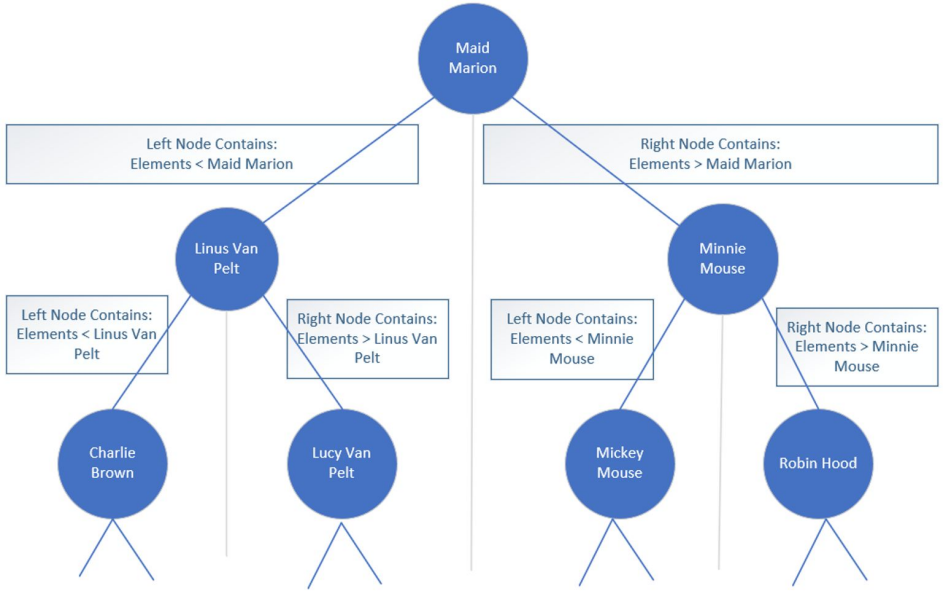
The **right** node and its children are elements that are **greater than** the parent node.



TreeSet

Instead of looking through all the elements in the collection to locate a match, this allows the tree to be quickly traversed, each node a simple decision point.

The main point is the tree remains balanced as elements are added.

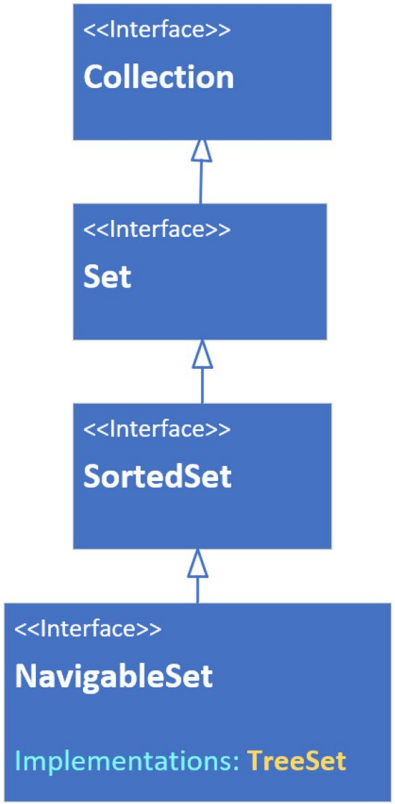


The TreeSet interface hierarchy

A TreeSet can be declared or passed to arguments typed with any of the interfaces shown on this slide.

This class is sorted, and implements the SortedSet interface, which has such methods as first, last, headSet and tailSet, and comparator.

This set also implements the NavigableSet Interface, so it has methods such as ceiling, floor, higher, lower, descendingSet and others.



The TreeSet relies on Comparable or Comparator methods

Elements which implement Comparable (said to have a natural order sort, like Strings and numbers) can be elements of a TreeSet.

If your elements don't implement Comparable, you must pass a Comparator to the constructor.

NavigableSet methods to get closest matches

All the methods shown on this slide take an element as an argument, and return an element in the set, the closest match to the element passed.

Element passed as the argument	**Result From Methods**			
	floor(E) (<=)	**lower(E) (<)**	**ceiling(E) (>=)**	**higher(E) (>)**
In Set	Matched Element	Next Element < Element or null if none found	Matched Element	Next Element > Element or null if none found
Not in Set	Next Element < Element	Next Element < Element or null if none found	Next Element > Element	Next Element > Element or null if none found

Getting subsets from a TreeSet.

All three methods, headSet, tailSet and subSet return a subset of elements, backed by the original set.

sub set methods	inclusive	description

`headSet(E toElement)`
 	
`headSet(E toElement, boolean inclusive)`
	toElement is exclusive if not specified	returns all elements less than the passed toElement (unless inclusive is specifically included).

`tailSet(E fromElement)`
 	
`tailSet(E toElement, boolean inclusive)`
	fromElement is inclusive if not specified	returns all elements greater than or equal to the fromElement (unless inclusive is specifically included).

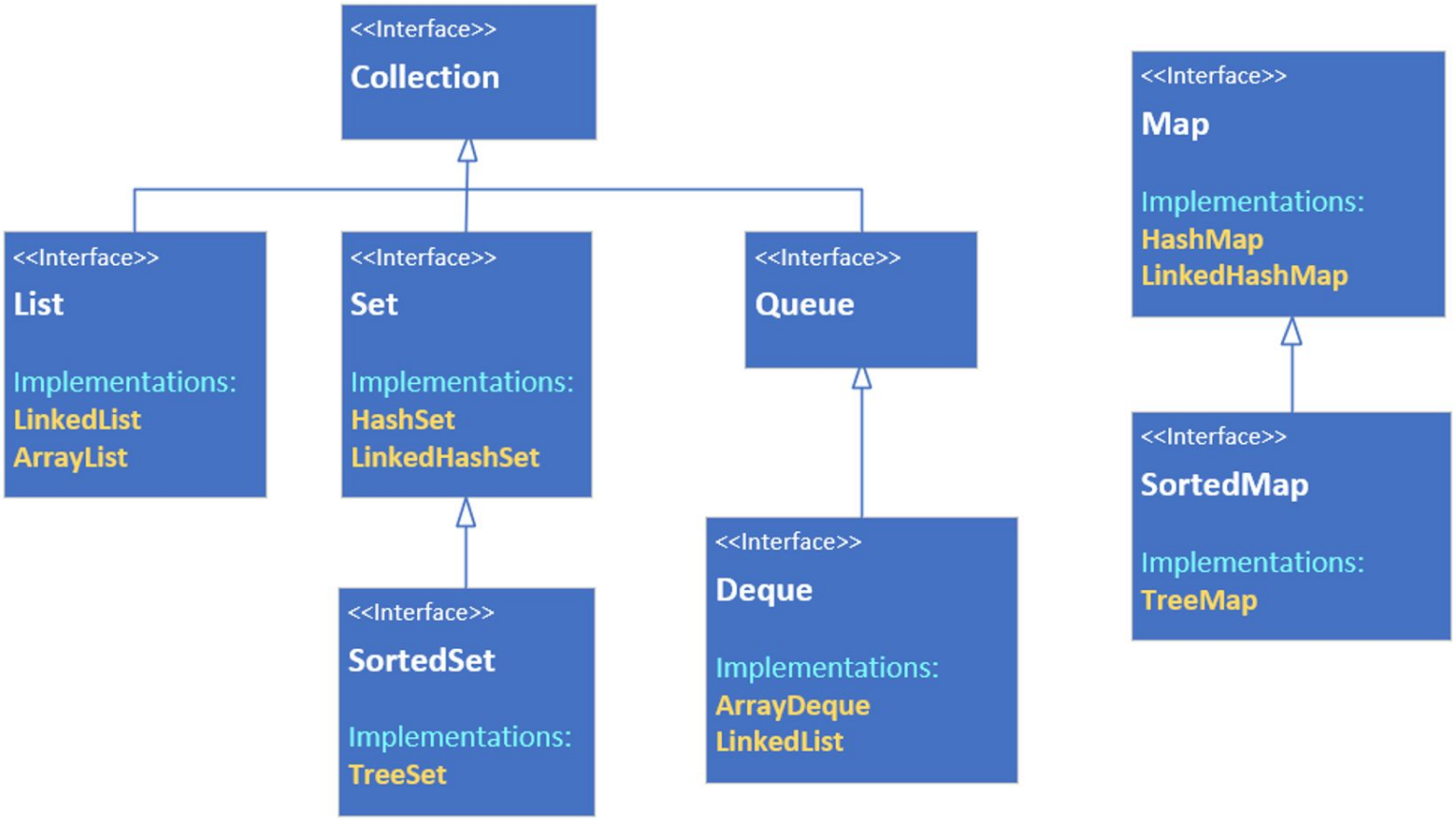
`subSet(E fromElement, E toElement)`
 	
`subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)`
	fromElement is inclusive if not specified,
	
	toElement is exclusive if not specified
		returns elements greater than or equal to fromElement and less than toElement.

When would you use a TreeSet?

The TreeSet does offer many advantages, in terms of built-in functionality over the other two Set implementations, but it does come at a higher cost.

If your number of elements is not large, or you want a collection that's sorted, and continuously re-sorted as you add and remove elements, and that shouldn't contain duplicate elements, the TreeSet is a good alternative to the ArrayList.

The Map Interface, why is it different?



The Map Interface, why is it different?

A map in the collections framework is another data structure.

Although it's still a grouping of elements, it's different, because elements are stored with keyed references.

This means a Map requires two type arguments, as you can see on this slide, where I'm showing the root interface, Collection, compared to the Map interface.

Collection Interface	Map Interface
`interface Collection<E> extends Iterable<E>`	`interface Map<K, V>`

The Map has K for it's key type, and V for the value type.

As with any generic classes, the only restriction on these types is, they must be reference types, and not primitives.

Map characteristics

A Java Map can't contain duplicate keys.

Each key can only map to a single value.

The Java classes that implement the map interface, the **HashMap**, the **LinkedHashMap**, and the **TreeMap**.

The HashMap is unordered, the LinkedHashMap is ordered by insertion order, and the TreeMap is a sorted map.