Sets

Before learning about the set data structure, it is prudent to explore the concept of sets in mathematics. This way you understand the mathematical collection from which this data structure originates.

In mathematics, sets are unordered collections without duplicates. Objects that are contained inside sets are called elements of that set.

When writing out sets mathematically we can use the following notation (called the roster method):

$$\{apple, orange, peaches, grapes\}$$
$$\{1, 2, 3, 4, 5\}$$

Notice that the two following sets are equal under our definition:

$$\{1, 3, 2\} = \{2, 1, 3\}$$

Two sets A and B are equal if the elements in A are all contained in set B and all elements of set B are contained in set A. Also notice that each element may be contained only once in the set. We cannot have the following set:

$$\{orange, orange\}$$

There are some other useful definitions to know about sets. A set A is called a subset of set B if all elements in set A are also in set B. We use the following notation to denote subset:

$$A \subseteq B$$

An alternative definition of set equality is $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.
Another piece of notation from set theory is the element of symbol. We use this to denote when one element is contained in a set. So the following is a true statement:

$$apple \in \{apple, peach, orange\}$$

Sets also have a few handy operations:

Set Union

Given two sets A and B, the union of those sets is defined as the set containing all elements in either A or B. Union is denoted $A \cup B$.

$$\{apple, orange, peach\} \cup \{apple, tomato, carrot\} = \{apple, orange, peach, tomato, carrot\}$$

Set Intersection

Given two sets A and B, the intersection of those sets is defined as the set containing all elements in both A and B. Intersection is denoted $A \cap B$.

$$\{apple, orange, peach\} \cap \{apple, tomato, carrot\} = \{apple\}$$

Set Difference

Given two sets A and B, the set difference of those sets is defined as the set containing all elements in A but not in B. Set difference is denoted $A - B$.

$$\{apple, orange, peach\} - \{apple, tomato, carrot\} = \{orange, peach\}$$

Set theory goes much deeper than this but a brief introduction to these concepts will greatly aid in your understanding of not only the data structures below but more advanced concepts based off Set Theory such as Inclusion-Exclusion, the Disjoint Sets data structure, and brute force concepts enumerating sets.

HashSets

This class is a way of implementing the unordered form of Sets. HashSet doesn't have as many fancy functions as TreeSet but makes up for it by being faster in most cases.

The concept of hashing is a standard concept taught in college level computer science courses. The idea is to take an object of some kind and encode it into an integer. That integer represents some element in a table (usually implemented by an array). This is a great idea until you end up with two objects with the same hash value. Such an occurrence is called a collision. If you are interested in this concept further, including how collisions are resolved, go look it up online. There are many resources on how this concept is implemented.

In order to go into a hash set the object must override both the **hashcode** and **equals** method from object. The hashcode generate an index for our object and equals must tests if two objects of the same type are indeed perfectly equal. (Equals is used to resolve collisions) If the hashing function is good enough, then collections will occur infrequently yielding $O(1)$ performance. If you use Integer, Long, String, or most build in java classes, these functions will already be implemented for you.

HashSets are a nice alternative if you don't want to sort the data you are hashing. They are also a nice choice for strings as many of the comparisons only happen on the **hashcode** and not the **equals** method.

Useful methods

```
hs.add(val);     // Adds the specified value to the HashSet
hs.remove(val); // Removes the specified value from the HashSet
hs.contains(val); // Returns if the specified value is in the HashSet
```

Maps

Similar to TreeSet and HashSet there are two types of Maps in java that are useful, TreeMap and HashMap. These two types of data structures, Sets and Maps, are very much related.
Maps are like more general arrays. They take in an object called a key and spit out a value. Unlike arrays, the key can be any object supported by the map.

Using generics for defining a map is a little different as you must specify both the key and the value's type for the map. Here is example usage for TreeMap and HashMap in java:

```
TreeMap<String, Integer> map1 = new TreeMap<String, Integer>();
HashMap<String, Integer> map2 = new HashMap<String, Integer>();
```

In Java 7 and later, we can use the same shortcut to cut out some of that typing:

```
TreeMap<String, Integer> map1 = new TreeMap<>();
HashMap<String, Integer> map2 = new HashMap<>();
```

In this example, our key is a String and it maps to an Integer. So we could use map as a way of mapping someone's last name to their age. There is a helpful method to aid in that mapping called **put**.

```
map.put("John", 16);
```

In this example, we take the string john and assign the value 16 to that position in the map. If we use array syntax (not allowed by Java) the code would look something like this:

```
map["John"] = 16
```

You can also retrieve the mapping using the **get** method:

```
int value = map.get("John");
```

At the end of this call, value would contain 16.
Maps serve as a way of creating associations between two types of data.

The difference between TreeMap and HashMap are similar to the differences between TreeSet and HashSet. Many of the same functions that exist for the Set version also exist for maps operating on the keys. (For example, **lower()** becomes **lowerKey()**) So you can take advantage of many of the same benefits of the set structures in the maps.