# HashSets

**Length:** 15 minutes
**Pre-Requisite Knowledge:** data structures, classes, interfaces (optional)
**Additional Resources:**

GitHub Repository
Interfaces in Java
Online Java Compiler

## Warm Up (2 min)

Discuss:

*Imagine that you have a program that tracks attendees who check in for an event using a list. One individual mistakenly checks in twice - now you have a duplicate entry!*

*How can we ensure there are no duplicates at all?*

*How might we solve this problem with an `ArrayList`?*

In Java, we have a collection type called a `Set`, which removes duplicates automatically!

## Review: Interfaces (1 min)

What is an **interface**?

- A special class that is **abstract**, meaning it cannot be instantiated directly.
- Used like a template for other classes, which **implements** the interface.
- **All** abstract methods in an interface must be defined in the class implementing the interface.

## The `HashSet` Class (5 min)

`HashSet` is a class that implements the `Set` interface! A `Set` has **two** main features:

- Does **not** allow duplicates.
- Has **no** guaranteed order.

It's implemented using something called a **hash table**, which allows it to add, remove, and search the set *very* quickly!

Differences between lists and sets are pretty simple:

| **Features** | **ArrayList** | **HashedSet** |
|---|---|---|
| Duplicates | ✔ | ✘ |
| Order | ✔ | ✘ |
| Average Efficiency | $O(n)$ | $O(1)$ |

## Importing `HashSet` (1 min)

Before we can create a `HashSet` object, we need to **import** it from the Java `util` library. We can can do this in one of two ways:

```
import java.util.HashSet;
import java.util.Set; // Not always necessary.
```

**or**

```
import java.util.*; // Imports ALL of the classes from the util package.
```

## Creating a `HashSet` Object (3 min)

We create a `HashSet` object like any other class, using the `new` keyword.

Often programmers prefer to make the defined type `Set` and the actual type `HashSet`, to provide more flexibility if we ever want to change the actual type of the object to another `Set` implementation. However, this isn't necessary, despite being *common practice*.

```
Set<String> attendees = new HashSet<>();
// Adding items to the HashSet.
attendees.add("Angel");
attendees.add("Alice");
attendees.add("Bob");
attendees.add("Alice");

System.out.println(attendees);
```

Discuss:

*What do you think this code will output?*

```
/* Output:  */
[Bob, Angel, Alice]
```

There are **no duplicates**, and it **does not** maintain the order we entered the values in!

We can also instantiate a `HashSet` using a **list of items**.

```
List<String> attendees = Arrays.asList("Angel", "Alice", "Bob", "Alice");
Set<String> uniqueAttendees = new HashSet<>(attendees);

System.out.println("Original: " + attendees);
System.out.println("Unique: " + uniqueAttendees);
```

Discuss:

*What do you think the output will be?*

*How will the original list and set be different from one another?*

```
/* Output: */
Original: [Angel, Alice, Bob, Alice]
Unique: [Bob, Angel, Alice]
```

## `HashSet` Methods (1 min)

There are quite a few useful methods from the class as well.

```
uniqueAttendees.add("Derek");      // Adds Derek to the set.
uniqueAttendees.contains("Angel"); // true
uniqueAttendees.remove("Bob");     // Removes Bob from the HashSet.
uniqueAttendees.size();            // 2
```

# Limitations and Alternatives (2 min)

`HashSets` have a few limitations, and should be chosen *based on your program's needs*.

`HashSets` are unordered due to its hashing, which organizes the values **arbitrarily** for **efficient searching**.

> If you need to **retain the order** the items are added, consider using the `LinkedHashSet` class instead, which retains the insertion order.

Only one `null` value is allowed, due to no values being able to be duplicates of one another.

> If you need **duplicate values**, or a default value (such as `null`) an implementation of the `List` class may be necessary.

# Conclusions (1 min)

`HashSet` provides efficient adding, removing, and lookup algorithms, only has unique elements, and is unordered.

Use Cases: removing duplicates, very quick operations - $O(n)$ efficiency on average.
Real-Life Examples:

- Tracking unique visitors of a website.
- Filtering and deduplicating input, such as the tags on a social media post.
- Fast intersection and union operations: find students enrolled in both algebra and CS, find all unique skills in a job applicant pool.

# Check for Understanding (4 min)

*What happens if I add the same element twice to a `HashSet`?*

*Which `Set` would I use if I wanted to maintain insertion order?*

*Other than the examples described in this lecture, what's one real-world example where you'd use a `Set` instead of a `List`?*

**To Do:**

- [x] ~~Lesson Guide~~
- [x] ~~Slide Set Presentation~~
- [ ] Code Demo Repository
- [x] ~~Make Handouts (?)~~
    - Could have QR codes to the repository.
- [ ] Print Resume (like 5?)