Data Structures

Iterable

In Java, `Iterable` is an interface that allows an object to be the target of the **for-each loop** (also known as the enhanced for loop). Any class that implements the `Iterable` interface promises to provide an `Iterator`, which is the object that actually steps through the collection's elements one by one.

The Core Concept

Think of `Iterable` as a contract that says, "I can be looped over." When you write a for-each loop like this:

```
for (Element e : myIterableObject)
{
    // do something with e
}
```

The Java compiler automatically uses the `Iterable` interface to get an `Iterator` from `myIterableObject`. It then uses that `Iterator` to go through each element until it runs out.

Iterable vs. Iterator

These two interfaces often confuse students, but their roles are distinct:

- `Iterable`: This is a collection-level interface. A class like `ArrayList` or `LinkedList` is `Iterable`. It has a single method, `iterator()`, which returns an `Iterator` object. It's the "factory" that produces the `Iterator`.
- `Iterator`: This is the object that does the actual work of iterating. It has three main methods:
  - `hasNext()`: Returns `true` if there are more elements to iterate over.
  - `next()`: Returns the next element in the sequence.
  - `remove()`: (Optional) Removes the last element returned by `next()`.

So, the `Iterable` object is the container, and the `Iterator` is the "pointer" that moves through the container.

Why is it Useful?

The `Iterable` interface is a powerful part of Java's Collections Framework because it provides a consistent, simple way to loop over different types of collections. Whether you have a `List`, a `Set`, or a custom collection, as long as it's

`Iterable`, you can use the same for-each loop syntax. This simplifies your code and makes it more readable, abstracting away the underlying implementation details of how the elements are stored.

"Authored" by Gemini, with some editing by Alyce Brady, 2 October, 2025.