

The background of the slide is a reproduction of a painting in the style of J.M.W. Turner or Vincent van Gogh. It depicts a landscape with a vibrant yellow field in the foreground, a blue sky filled with numerous dark birds in flight, and a winding path or stream in the middle ground. The brushstrokes are thick and expressive, characteristic of Impressionism.

# J14 Collections

The collections framework  
Common collection types  
Iterator and Stream interfaces  
Ordering collections



# The Collections Framework

(in `java.util`)

- Interfaces
  - Implementation-agnostic interfaces for collections
- Implementations
  - Concrete implementations
- Algorithms
  - Searching, sorting, etc.

Using the framework saves writing your own: better performance, fewer bugs, less work, etc.

# The Collection Interface

- Basic operators
  - `size()`, `isEmpty()`, `contains()`, `add()`, `remove()`
- Traversal
  - Iterators (abbreviated for loop syntax), `forEach()`.
- Bulk operators
  - `containsAll()`, `addAll()`, `removeAll()`, `retainAll()`, `clear()`
- Array operators
  - convert to and from arrays

# Collection Types

- Primary collection types:
  - **Set** (no duplicates, mathematical set)
  - **List** (ordered elements)
  - **Queue** (ordered sequence with restrictions)
  - **Map** (<key, value> pairs)
- Each collection type is defined as an interface
  - You need to choose a concrete class to instantiate
  - Your choice will depend on your needs

# Concrete Collection Types

	<i>Implemented Using</i>				
<i>Interfaces</i>	Hash table	Resizable array	Tree	Linked list	Hash table + linked list
<b>Set</b>	HashSet		TreeSet		LinkedHashSet
<b>List</b>		ArrayList		LinkedList	
<b>Queue</b>		ArrayDeque		LinkedList	
<b>Map</b>	HashMap		TreeMap		LinkedHashMap

Based on table from <http://docs.oracle.com/javase/tutorial/collections/implementations/index.html>

# Four Commonly Used Collection Types

- HashSet implements a **set** as a hash table
  - Makes no ordering guarantees
- ArrayList implements a **list** using an array
  - Very fast access
- HashMap implements a **map** using a hash table
  - Makes no ordering guarantees
- LinkedList implements a **queue** or **list** using a linked list
  - First-in-first-out (FIFO) queue ordering

# Iterable<T> interface

Collections implement the `Iterable<T>` interface, which enables use of the “For-Each loop”:

```
for (var t : things) {  
    System.out.println(t);  
}
```

and also a `forEach` method to apply lambda expression:

```
things.forEach(t -> System.out.println(t));
```

# Stream<T> Interface

Collections can be accessed as a stream via the `stream()` method, enabling a more **functional programming** style:

```
List<Integer> list = List.of(1, 2, 3, 4, 5); // immutable list!
var count = list.stream()
    .filter(x -> x > 2)
    .count();
var nList = list.stream()
    .filter(x -> x > 2)
    .map(x -> Integer.toString(x + 2))
    .toList(); // immutable, otherwise collect(...)
```



# Ordering Collections

The Comparable interface defines a 'natural' ordering for all instances of a given type, T:

```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

The return value is either negative, 0, or positive depending if the receiver comes before, equal, or after the argument, o.

The Comparator *functional* interface allows a type T to be ordered in ad-hoc ways:

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

# java.util.Collections

Some useful static methods for collections:

- `sort`, `min`, `max`, `reverse`, `frequency`, `addAll`

`List` also has a `sort` instance method:

- When provided with `null` it uses the natural order of elements (given by `Comparable`)
- Can use bespoke ordering when provided a lambda expression (`Comparator` functional interface):

```
(T a, T b) -> { return <expression>; }
```

# Josh Bloch Item 25: Prefer lists to arrays

## Why?

- Arrays are *covariant*, Generics are *invariant*
  - if A **extends** B, then A[] is a subclass of B[]
  - but List<A> has no relationship to List<B>

```
// Fails at runtime!
```

```
Object[] array = new Long[1];  
objectArray[0] = "I don't fit in"; // Throws ArrayStoreException
```

```
// Won't compile!
```

```
List<Object> list = new ArrayList<Long>(); // Incompatible types  
list.add("I don't fit in");
```