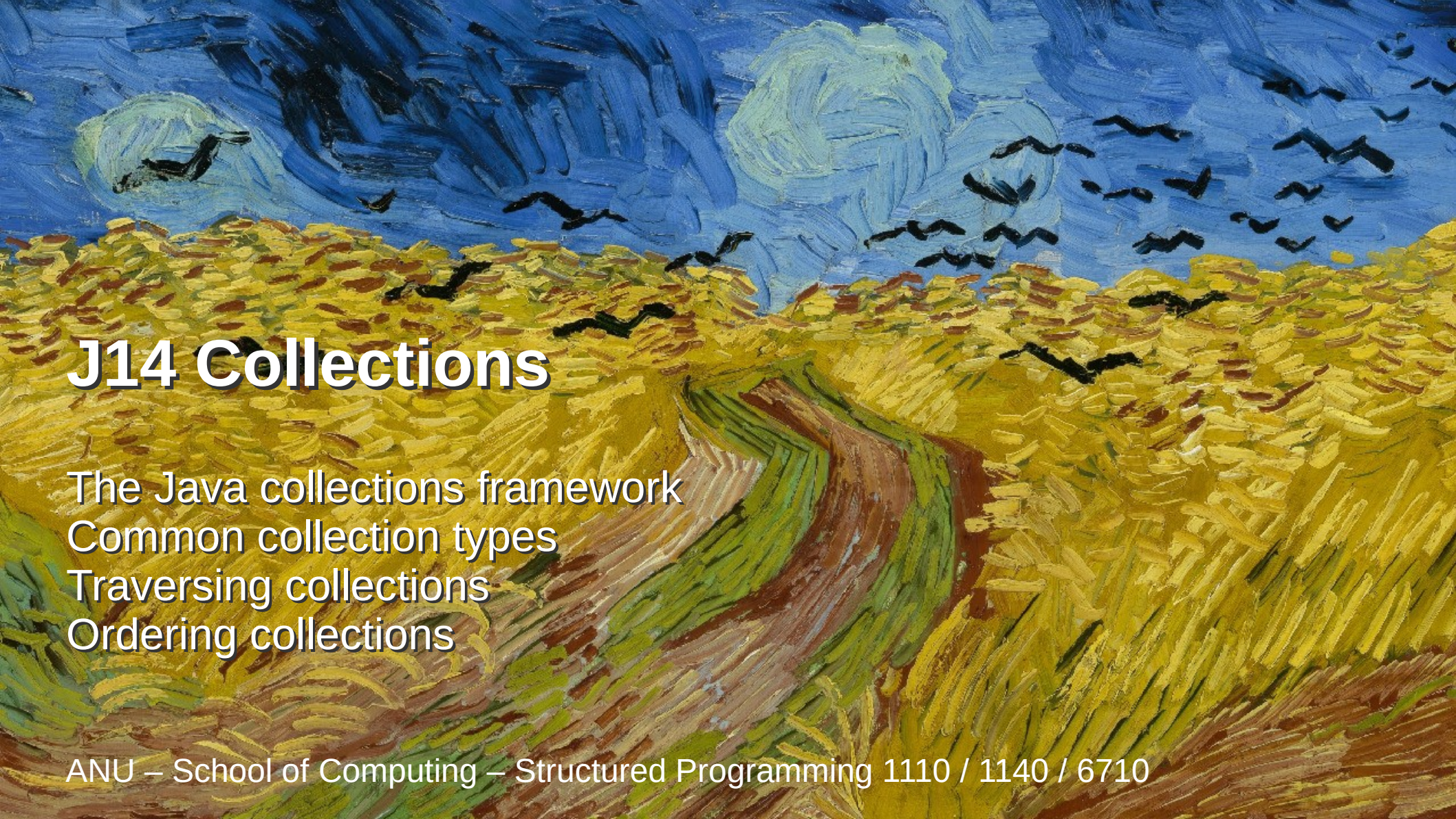

J14 Collections

The Java collections framework
Common collection types
Traversing collections
Ordering collections

The Java collections framework

- Defined in the `java.util` package.
- 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.

Collection interfaces

- Basic operations
  - `size()`, `isEmpty()`, contains(…), add(…), remove(…)
  - Note: `contains` (and `add` in `Set`) uses type-specific `equals`.
- Traversal
  - Iterators (abbreviated for loop syntax), stream interface.
- Bulk operations
  - `containsAll(…)`, addAll(…), removeAll(…), `clear()`
- Conversion to and from arrays.
- Note: Default collection types are modifiable, but some methods create immutable collections.

Collection types

- Primary collection types:
  - **Set** (no duplicates, mathematical set)
  - **List** (ordered sequence of variable length)
  - **Queue** (ordered sequence with restrictions)
  - **Map** (set of <key, value> pairs, indexable on key)
- 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

	Implemented Using				
Interfaces	**Hash table**	**Resizable array**	**Tree**	**Linked list**	**Hash table + linked list**
Set	`HashSet`		`TreeSet`		`LinkedHashSet`
List		`ArrayList`		`LinkedList`	
Queue		`ArrayDeque`		`LinkedList`	
Map	`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
  - Fast (constant-time) 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

The `Iterable<T>` interface

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

```
for (String s : aListOfStrings) {
    System.out.println(s);
}
```

- The `forEach` method applies a function to each element:

```
aList.forEach(x -> System.out.println(x));
```

Ordering elements

- The `Comparable` interface defines the “natural” ordering of a given type:

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

- `a.compareTo(b)` returns < 0 if `a` is ordered before `b`, > 0 if `b` before `a`, and 0 if `a` and `b` are equal.
- The `Comparator` *functional* interface defines an ad-hoc ordering for type `T`:

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

The `java.util.Collections` class

- Container class for static generic methods for collections:
  - `sort`, `min`, `max`, `reverse`, `frequency`, `addAll`
- The `List` interface also has a `sort(Comparator)` instance method:
  - Sorts the list according to the order defined by the provided Comparator implementation.
  - If null, sorts according to the element type's natural order.

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");
```