LEARN PROGRAMMING IN JAVA™

Learning OO Java™ as a Beginner – Collections

COURSE AGENDA

- What is an Object
- The Basics
- Operators
- Control Flow
- Implementation Hiding
- Reuse
- Interfaces
- Polymorphism

- Collections
- Functional Programming
- Stream
- Exceptions
- Enums

INSTRUCTOR – HUGO SCAVINO

- 30 Years of IT Experience
- Using Java since the beta
- Taught Java and OOP in USA, UK, and France to Fortune 500
- Senior Software Architect with
  - DTCC
  - Penske
  - HSBC
  - Government Agencies

https://www.linkedin.com/in/hugoscavino/

REQUIRED SOFTWARE

- JDK™
  - JDK 8 or Newer (Open Source, Amazon, or Oracle)
  - The beginner course focuses on core language constructs; feel free to use 11, 17, 21, etc.

- IDE
  - IntelliJ Community Edition (Free) or Enterprise (Paid)
  - While you can use any IDE with Java, the course and labs are explicitly made with IntelliJ in mind

RECOMMENDED TEXTS

- Java 5 Book
  - Thinking in Java – 4th Edition - Free – PDF - GitHub
  - Thinking in Java – 4th Edition – Hard Cover - Amazon

- Recommended Java 8 Book
  - Bruce Eckel on Java 8
    - Contains references to newer Java syntax
- #1 Best Seller in Beginner's Guide to Java Programming
  - Head First Java: A Brain-Friendly Guide 3rd Edition

TOPIC DESCRIPTION

Collections: (Java 8 above) a comprehensive set of classes and interfaces designed to handle collections of data. Providing a standard way to manage groups of objects and includes utilities for data manipulation like sorting, searching, and iterating. With Java 8 the inclusion of functional programming features have made it more versatile.

Core Concepts:

Interfaces: Define the abstract data types, where Collection is the root for most.
Implementations: Concrete classes that implement the above `ArrayList` and `Set`.
Utility Classes: Provide static methods for common usages and manipulation scenarios.

ENHANCEMENTS SINCE JAVA 8

Stream API: Allows for functional-style operations on collections and maps. The concepts are very similar to those found in Angular and React (TypeScript) programming. Enables efficient (albeit less easy to read) data processing using lambdas and method references.

Common Operations:

- Filter: `collection.stream().filter(x -> condition)`
- Map: `collection.stream().map(x -> transform)`
- Reduce: `collection.stream().reduce(accumalator)`
- Collect: `collection.stream().collect(Collectors.toList())`

KEY ADVANTAGES

Functional Programming: Simplified and expressive operations using lambda expressions and streams.
Immutability: Simplified creation of immutable collections.
Efficiency: Improved performance and cleaner code through new utilities.
Parallelism: Easy parallel processing with parallelStream().

WHAT NOT TO DO

Typeless Collections

You may encounter legacy code with `ArrayList` that does not type check when inserting or removing. DON'T USE THE OLD FRAMEWORK.

```
// DO NOT DO THIS ANYMORE!
// WARNING: RAW USE OF LIST
ArrayList list = new ArrayList();

list.add(new Apple());
list.add(new Apple());
list.add(new Orange()); // YIKES!
list.add(new Apple());

for (Object o : list) {
    Apple apple = (Apple)o; /// OH NO!!!!!
    apple.slice();
}
```

`ClassCastException: class ... Orange cannot be cast to class … Apple`

If you are stuck with legacy code, create a wrapper ("Façade Pattern") around the old interface and use types!

COMPILE TIME CHECKING EXAMPLE

```
// DO THIS INSTEAD
// List ONLY accepts Apple Type
List<Apple> list = new ArrayList<>();

list.add(new Apple());
list.add(new Apple());
list.add(new Orange()); // Compile Time!
list.add(new Apple());

for (Apple apple : list) {
    apple.slice(); // No Need to case object
}
```

When we declare the Type from the beginning the compiler will prevent mis-matched object in our collection.

SUBTYPES AND INTERFACES WORK!

```
public static class Apple{  6 usages  1 inheritor  new *
    public Apple() { System.out.println("I am an Apple"); }
    public void slice() { System.out.println("Apple Slice"); }

}
 public static class GrannyApple extends Apple{  no usages  new *
     public GrannyApple() { System.out.println("I am an Granny Apple"); }
 }

  List<Apple> list = new ArrayList<>();

  list.add(new Apple());
  list.add(new Apple());
  list.add(new GrannyApple());
```

YOUR LIFE IS ARRAYLIST AND MAP

Collection: a sequence of individual elements with one or more rules applied to them.

List must hold the elements in the way they were inserted.
Set cannot have duplicate elements, and
Queue produces the elements in the order determined by a queuing discipline

Map: a group of key-value object pairs that looks up a value using a key.

An ArrayList looks up an object using a number, so in a sense it associates numbers to objects. A map looks up an object using another object. Maps are powerful programming tools.

COLLECTION UTILS TO THE RESCUE

Adding Groups of Elements:

```
Collection<SafeCollection.Apple> apples = new ArrayList<>(
        Arrays.asList(
                new SafeCollection.Apple(),
                new SafeCollection.Apple(),
                new SafeCollection.Apple(),
                new SafeCollection.GrannyApple()));
```

Using the `Arrays.asList(...)` utility method

COLLECTION UTILS TO THE RESCUE

Adding Groups of Elements: addAll()

```
SafeCollection.Apple[] appleArray = {new SafeCollection.Apple(), new SafeCollection.GrannyApple()};
apples.addAll(Arrays.asList(appleArray));
System.out.println("Apples in collection after allAll() : " + apples.size());
```

- Add to the Collection using an Array

COLLECTION UTILS TO THE RESCUE

Adding Groups of Elements: Collections.addAll()

```
// Runs faster, but cannot use this method to construct the collection
Collections.addAll(apples, new SafeCollection.GrannyApple());
System.out.println("Apples in collection after Collections.allAll() : " + apples.size());
```

- Add Apples using the `Collections.allAll()` utility function.
- The apple collection need to be created first!

LIST INTERFACE

The `List` interface maintains elements in a particular sequence. It adds several methods to `Collection` that allow insertion and removal of elements in the middle of a `List`.

`ArrayList`, excels at randomly accessing elements, however, is slower when inserting and removing elements in the middle.

`LinkedList`, provides optimal sequential access, with inexpensive insertions and deletions in the middle of the List. `LinkedList` is slower for random access, however, has a larger feature set compared to `ArrayList`.

LIST EXAMPLE

```
final int count = animals.size();
System.out.println("Animals in List [" + count + "]");

boolean fidoFound = animals.contains(fido);
System.out.println("Is Fido in List? [" + fidoFound + "]");

boolean removed = animals.remove(dawg);
System.out.println("Was dawg removed [" + removed + "]");
System.out.println("Animals in List [" + animals.size() + "]");
```

Popular methods

- `add()`
- `size()`
- `contains()`
- `remove()`

ITERATING OVER A LIST

```java
// Step 1: Setting and initializing a variable
// as per syntax of while loop
int val = 0;

// Step 2: Condition
// Till our counter variable is lesser than size of
// ArrayList
while (animals.size() > val) {
    System.out.println(animals.get(val));
    // Step 3: Terminating condition by incrementing
    // our counter in each iteration
    val++ ;
}
```

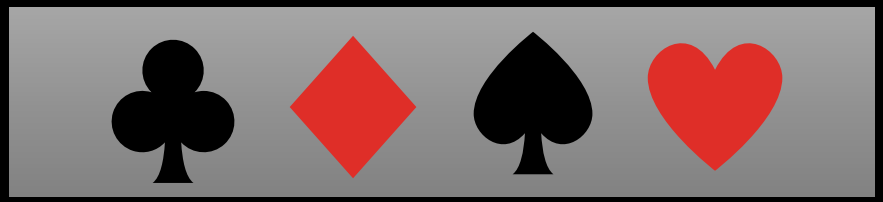
```java
// Iterating using for loop
for (int i = 0; i < animals.size(); i++){
    // Printing and display the elements in ArrayList
    System.out.print(animals.get(i) + " ");
}
```

SET

`Set` is a collection:

Does not allow duplicate elements: A `Set` contains only unique elements. If you try to add a duplicate, the operation will fail or have no effect.

`Set` has no defined order. The elements in a `Set` are generally unordered, meaning their order of iteration is not guaranteed unless you use a specific implementation like `LinkedHashSet`.



SET IMPLEMENTATIONS

HashSet

Uses a hash table for storage. Allows null elements. Does not maintain insertion order. Fast for operations like add, remove, and contains).

LinkedHashSet

A subclass of HashSet. Maintains insertion order. Slightly slower than HashSet because it uses a linked list internally.

TreeSet

Implements the Navigable Set interface, based on a Red-Black Tree.

SET NO DUPLICATES

```
Set<Card> suits = new HashSet<>();
suits.add(new Card(Values.ACE, new Diamonds()));
suits.add(new Card(Values.ACE, new Spades()));
suits.add(new Card(Values.ACE, new Clubs()));
// ignored b/c of equals()
suits.add(new Card(Values.ACE, new Clubs()));
suits.add(new Card(Values.ACE, new Hearts()));

for (Card c : suits) {
    System.out.println("Card : " + c.getValue() +
}
```

COMMON METHODS AND USE CASES

`add(Object o):` Adds the specified element if it is not already in the set.

`remove(Object o):` Removes the specified element if it exists in the set.

`contains(Object o):` Checks if the set contains the specified element.

`size():` Returns the number of elements in the set.

`isEmpty():` Checks if the set is empty.

`clear():` Removes all elements from the set.

To ensure unique elements.
When the order of elements is not critical, use a HashSet.
When ordered results are required, then `TreeSet` or `LinkedHashSet`.

MAP EXAMPLE

```
Map<String, SafeCollection.Apple> fruitInventory = new HashMap<>();

fruitInventory.put("A100", new SafeCollection.Apple());
fruitInventory.put("A200", new SafeCollection.Apple());
fruitInventory.put("A300", new SafeCollection.GrannyApple());

SafeCollection.Apple apple = fruitInventory.get("A100");
apple.slice();
```

- Defined the key to be a String and unique
- Above has three items in the `fruitInventory` Map
- Retrieve an `Apple` instance, using the key (“A100”) and the method get(Object o): V

MAP

`Map` is an object that maps keys to values. It is part of the `java.util` package and is designed to store data in key-value pairs. Each key in a `Map` must be unique, but values can be duplicated.

Key-Value Pairs: A Map associates a unique key with a value.

- Unique Keys: Keys in a Map must be unique. If a key is added that already exists, the old value associated with that key is replaced.
- Values May Be Duplicated: Unlike keys, values in a Map can be repeated.

Not a Collection: Map is not a subtype of Collection. It does not implement the Collection interface.

MAP

HashMap
Uses a hash table for storage. Does not maintain insertion order. Allows one null key and multiple null values.

LinkedHashMap A subclass of HashMap. Maintains insertion order. Slightly slower than HashMap because it uses a linked list internally.

MAP

TreeMap

Implements the `NavigableMap` interface. Maintains elements in sorted order (based on natural order or a custom comparator). Does not allow null keys (but allows null values).

Hashtable

Synchronized (thread-safe). Does not allow null keys or null values. This is a legacy class, use `ConcurrentHashMap` for thread-safe applications.

MAP EXAMPLE

```
Map<String, SafeCollection.Apple> fruitInventory = new HashMap<>();

fruitInventory.put("A100", new SafeCollection.Apple());
fruitInventory.put("A200", new SafeCollection.Apple());
fruitInventory.put("A300", new SafeCollection.GrannyApple());

SafeCollection.Apple apple = fruitInventory.get("A100");
apple.slice();
```

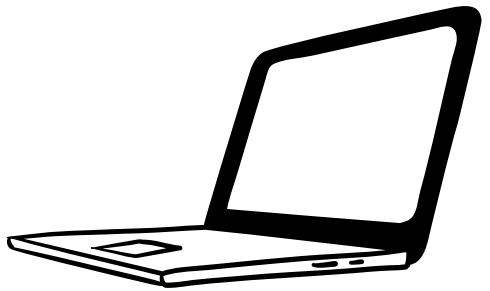
`Map` uses a hash table for storage. `Map` does not maintain insertion order. It allows one null key and multiple null values.

MAP COMMON METHODS

`put(K key, V value)`: Adds or updates a key-value pair.
`get(Object key)`: Retrieves the value associated with the key.
`remove(Object key)`: Removes the key-value pair for the key.
`containsKey(Object key)`: Checks if the map contains the key.
`containsValue(Object value)`: Checks if map contains the value.
`keySet()`: Returns a Set view of the keys.
`values()`: Returns a Collection view of the values.
`entrySet()`: Returns a Set view of the key-value pairs.

MAP USE CASES

- Associating unique identifiers (keys) with data (values), such as user IDs and names.
- Implementing lookup tables or dictionaries.
- Caching data for quick retrieval.
- Representing relationships or mappings between entities.



RECORD

Record is a class designed to represent immutable data concisely and boilerplate-free.

A record is a class with fields (called components), and the compiler automatically generates the following:

- Constructor
- Getters
- `equals()`, `hashCode()`, and `toString()`

RECORD EXAMPLE

```
public record Person(String fullName, int age) {}  no usages  new *
public record Order(int ID, String description) {}  no usages  new *
```

Limitations:

No Mutability: Immutable by design, cannot modify fields after creation.
No Custom Superclass: Cannot extend other classes; implicitly extend `java.lang.Record`.
Not Suitable for Complex Classes: Intended for simple, data-carrying objects, not classes with business logic or state management.

TOPIC SUMMARY

List

Set

Map

Record