GUVI | HCL
Skill Up. Level Up

Collections Framework

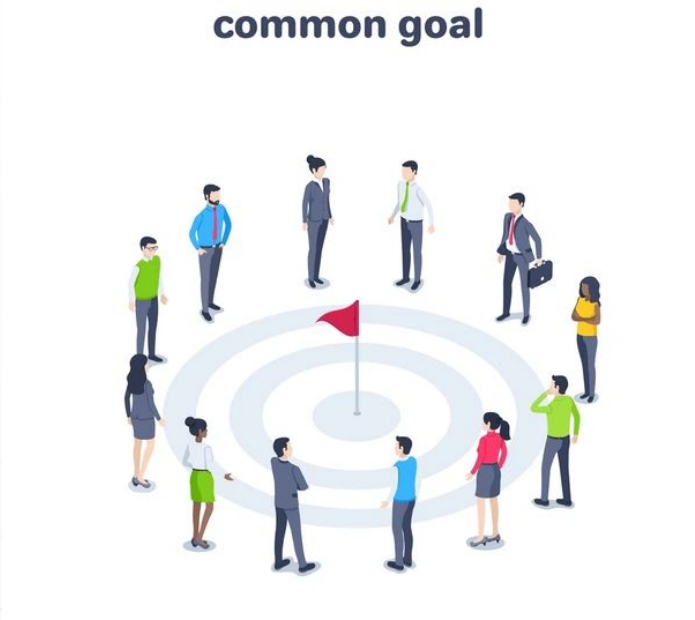
Programming Foundation

Presented by
Nikhil Nair

Website
www.guvi.com

Objectives

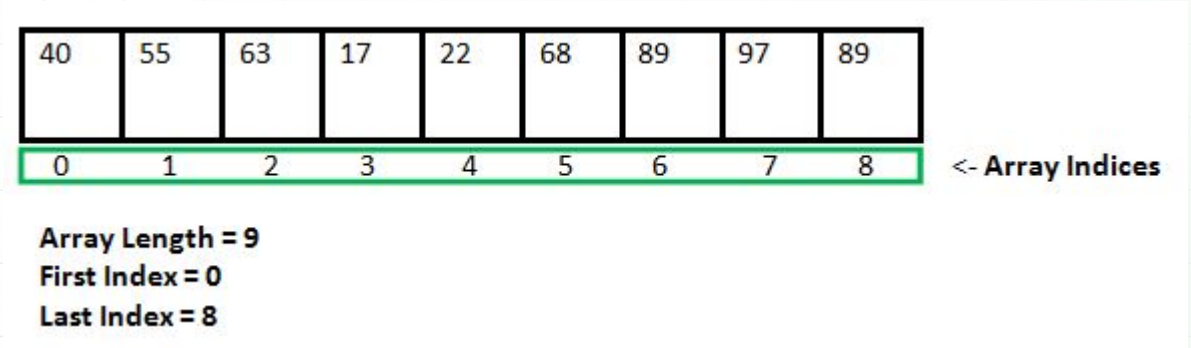
- What you will learn today
    - Challenges with existing set of data types
    - Collection Framework Overview
    - Introduction to
        - Lists
        - Sets
        - Maps
        - Queues



Collections: An Intro

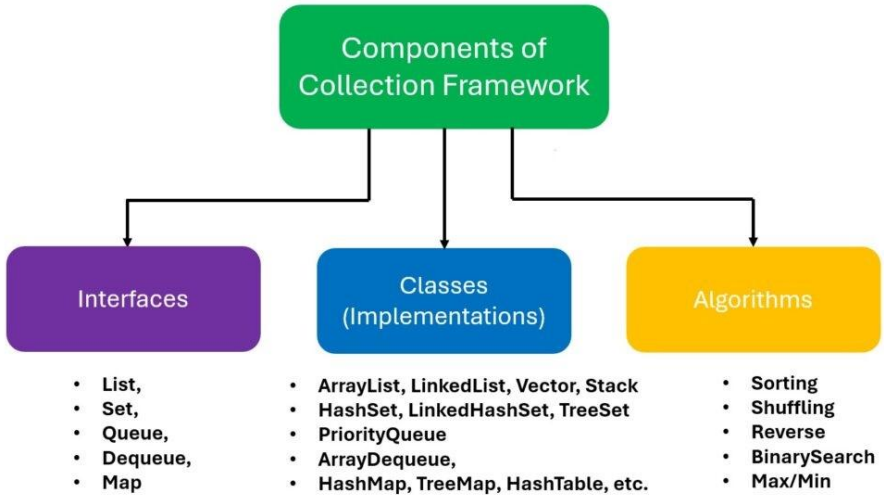
Why fixed arrays fall short

- Arrays have a fixed size that cannot change after creation
- Adding or removing elements requires copying and manual index updates
- Empty gaps or duplicate data often appear when size changes are manual
- Real programs need containers that adjust automatically as data grows or shrinks



What the Collections Framework solves

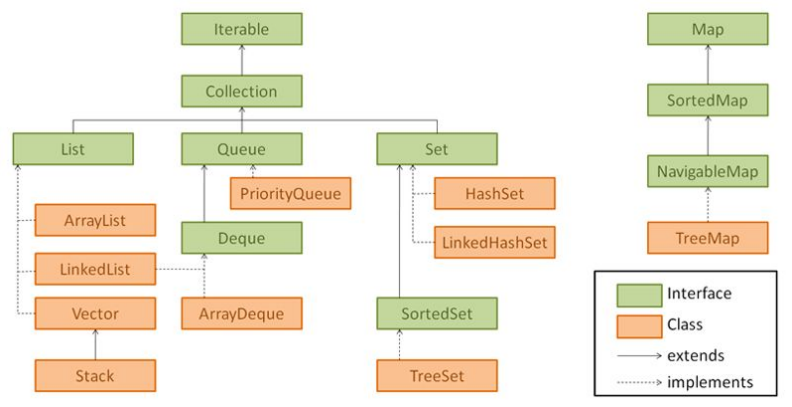
- Offers resizable, reusable structures for storing and retrieving grouped data
- Defines consistent methods like add, remove, and contains across types
- Removes need for manual array resizing or shifting during updates
- Improves code readability and reliability through unified design principles



Components of Collection Framework in Java

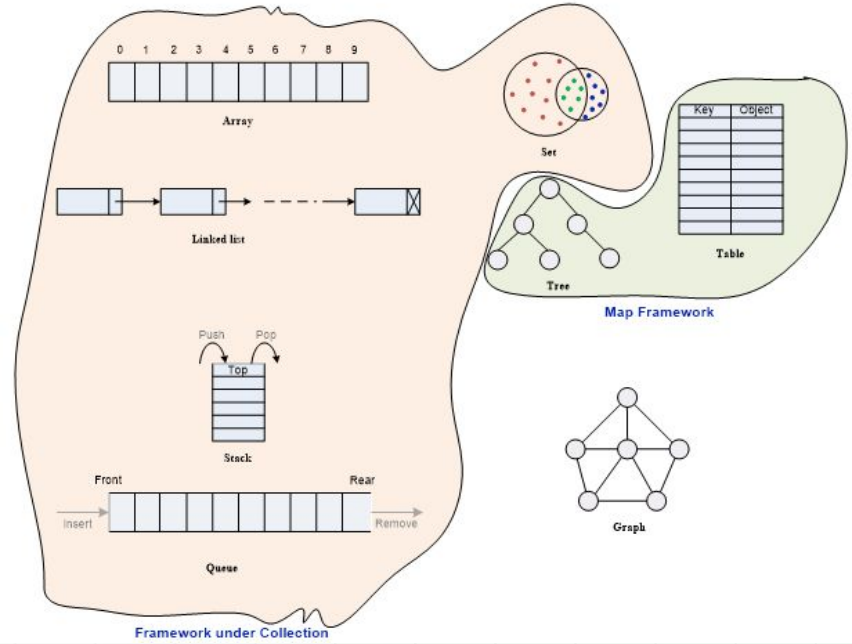
Collections hierarchy at a glance

- Core interfaces include Collection, List, Set, Map, and Queue
- Each defines how data is stored, ordered, or accessed
- Implementations like ArrayList or HashSet fulfill these rules in different ways
- The hierarchy helps choose the right structure for each programming problem



Core contracts of common collections

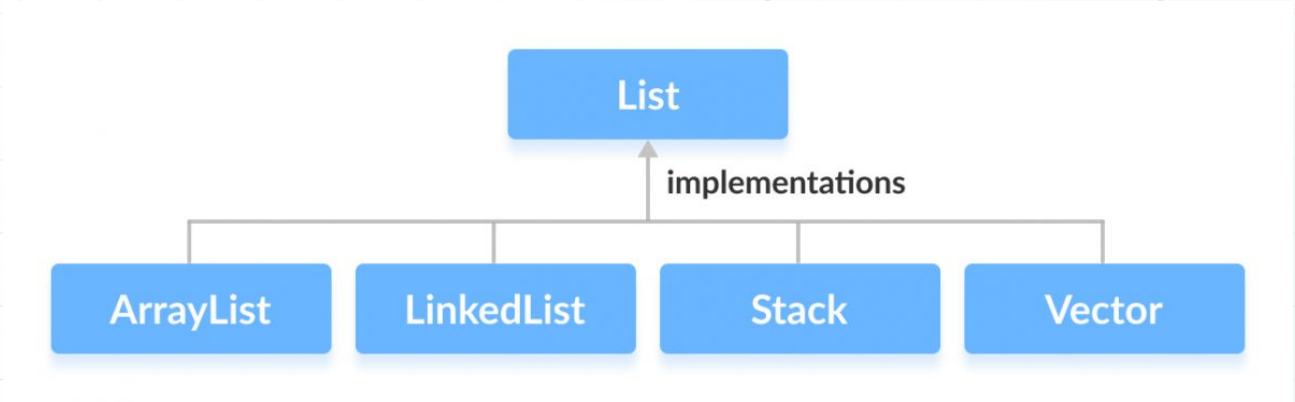
- Collection is a family of types that group elements together
- List defines ordered elements that are addressable by position
- Set defines unique elements where duplicates are not stored
- Map defines key to value pairs for fast lookups
- Queue defines first in first out style access to elements



Collections: Lists

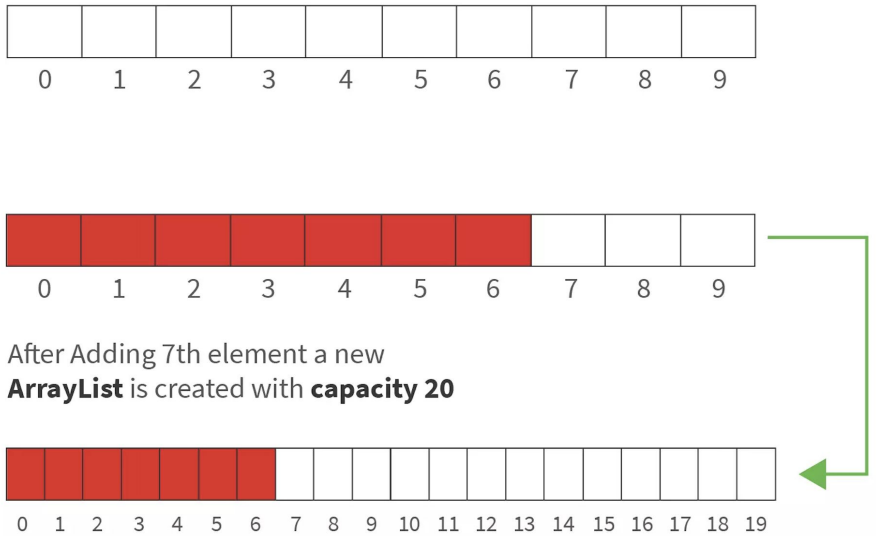
Lists as ordered collections

- A List stores elements in a specific order you control
- Elements are addressable by index starting from zero
- The same value can appear more than once if needed
- Choose List when order and positions matter for your task



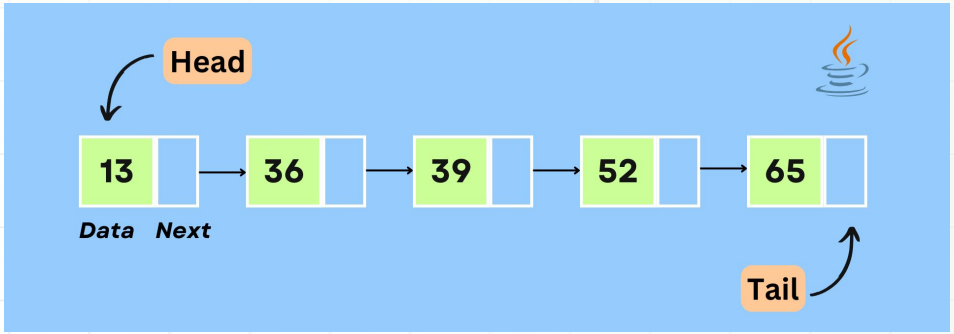
ArrayList as resizable array

- Backed by an array that grows automatically when needed
- Fast random reads by index for most everyday workloads
- Inserts or removals in the middle can be more expensive
- Prefer when frequent access by position is your main need



LinkedList as node based sequence

- Built from nodes that hold value and next links
- Efficient insertions and deletions at ends or known positions
- Random access by index is slower due to walking nodes
- Prefer when you add or remove items during traversal



Iterating safely through lists

- For each loop is simple for reading items in order
- Iterator supports safe removal during traversal when needed
- Do not modify a list during for each iteration
- Prefer index loop only when you truly need positions

```
import java.util.*;
List items = new ArrayList<>();
items.add("A"); items.add("B");
Iterator it = items.iterator();
while (it.hasNext()) {
  String s = it.next();
  if (s.equals("A")) {
    it.remove();
  }
}
```

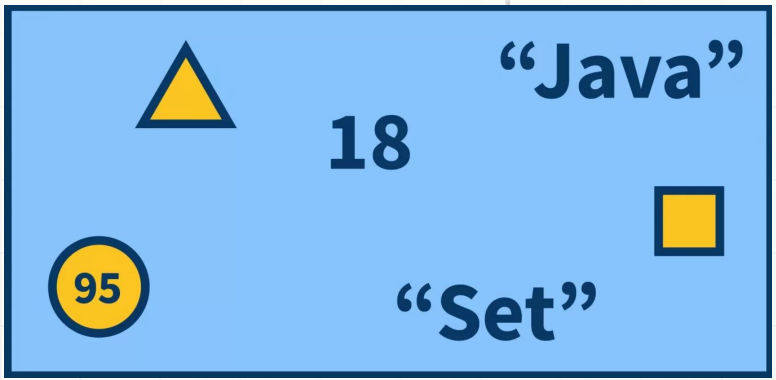
Activity: Shopping cart list

- Goal: build a small Cart that stores item names in insertion order and ignores exact duplicates
- Rules
    - addItem rules: reject null or blank strings; ignore duplicates; return boolean for success.
    - Remove during iteration: delete every item that starts with B in a safe manner
- Items to start: Start with: Apples, Bread, Banana, Milk, Bread
- Expected final cart: Apples, Milk

Collections: Sets

Sets for enforcing uniqueness

- A Set stores elements without keeping duplicate values
- Use Set when you must prevent repeated entries automatically
- HashSet does not guarantee iteration order across runs
- LinkedHashSet preserves insertion order while still skipping duplicates



HashSet and LinkedHashSet behavior

- HashSet does not guarantee iteration order across runs
- LinkedHashSet preserves insertion order while still skipping duplicates
- Both automatically ignore repeated adds of the same value

```
import java.util.*;
Set<String> a = new HashSet<>();
a.add("A"); a.add("B"); a.add("A");
System.out.println(a);
Set<String> b = new LinkedHashSet<>();
b.add("A"); b.add("B"); b.add("A");
System.out.println(b);
```

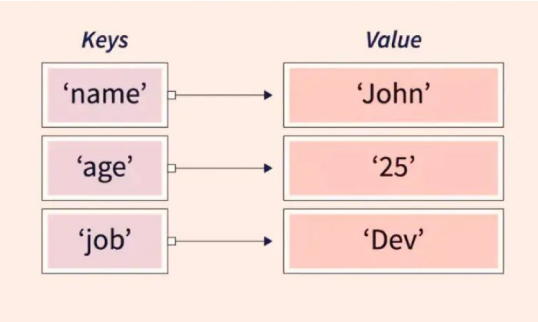
Activity: Normalizing names in a Set

- Given
  - Raw inputs with spaces, case differences, and a few invalid entries; **inputs**: " Asha ", "asha", "Ravi", "ravi", "Meena", "", null
- Build
  - A small interface named Normalizer with one method, apply
  - One concrete strategy (think - class) that trims and lowercases a given name
  - A LinkedHashSet to store normalized names while keeping first seen order
- Rules
  - Normalizer.apply returns null if name is null or empty after trim; only nonnull results are added to the LinkedHashSet
- Expected Outcome
  - **Final set**: ["asha", "ravi", "meena"]; **Skipped count**: 2

Collections: Maps

Maps as key value associations

- A Map stores pairs where each key links to one value
- Keys are used to read, update, or replace values quickly
- Keys are unique within a single Map instance
- Choose Map when you need fast named lookups for data



Basics of HashMap

- Fast lookups for keys without guaranteeing iteration order
- Adding a pair with an existing key replaces the old value
- Reading a missing key returns null rather than throwing an error
- Default choice when output order does not matter

```
import java.util.*;
Map m = new HashMap();
m.put("Latte", Integer.valueOf(120));
m.put("Tea", Integer.valueOf(80));
m.put("Latte", Integer.valueOf(140));
System.out.println(m.get("Latte"));
System.out.println(m.get("Mocha"));
for (Object k : m.keySet()) { System.out.print(k + " "); }
```

LinkedHashMap basics

- Preserves insertion order while keeping one value per key
- Replacing a value keeps the key in its original position
- Iteration follows the order in which entries were added
- Use when predictable output order is important for users

```
import java.util.*;
Map m = new LinkedHashMap();
m.put("Tea", Integer.valueOf(80));
m.put("Latte", Integer.valueOf(140));
m.put("Latte", Integer.valueOf(150));
for (Object k : m.keySet()) { System.out.print(k + " "); }
System.out.println();
System.out.println(((Integer)m.get("Latte")).intValue());
```

Map operations in practice

- Put adds or replaces a value for the same key
- Get reads a value for a key or returns null if missing
- Contains key checks existence without retrieving the value
- Remove deletes a key and its value from the map

```
import java.util.*;
Map m = new HashMap();
m.put("Tea", Integer.valueOf(80));
m.put("Tea", Integer.valueOf(90));
System.out.println(m.get("Tea"));
System.out.println(m.containsKey("Latte"));
m.remove("Tea");
System.out.println(m.get("Tea"));
```

Map operations in practice

- Put adds or replaces a value for the same key
- Get reads a value for a key or returns null if missing
- Contains key checks existence without retrieving the value
- Remove deletes a key and its value from the map

```
import java.util.*;
Map m = new HashMap();
m.put("Tea", Integer.valueOf(80));
m.put("Tea", Integer.valueOf(90));
System.out.println(m.get("Tea"));
System.out.println(m.containsKey("Latte"));
m.remove("Tea");
System.out.println(m.get("Tea"));
```

Activity: Cafe menu service

- Goal
  - Represent a cafe menu in Java using a HashMap and a LinkedHashMap
- Build
  - Create a class, MenuService, with two instance variables (type HashMap & LinkedHashMap)
  - Implement two methods: updatePrice(Map m, String item, Integer price) and total(Map m)
- Rules
  - updatePrice throws IllegalArgumentException if price is null or negative; if item is valid, use put to insert/update both maps
  - total sums all Integer values in the given map
- Task
  - Initialize both maps with products: Tea (price: 80) and Latte (140); update Latte's price to 150 and add Mocha (110)
  - Attempt an invalid update for Tea with a negative price and handle the exception at the call site with a one line message
  - Print both maps and the total from the LinkedHashMap

Collections: Queues

Queues as first in first out

- A Queue is a first-in-first out data structure; it adds elements at the tail and removes from the head
- It provides several methods, including:
    - Peek: reads the next item without removing it
    - Poll: removes and returns the next item or returns null if empty
- Use when tasks should be handled in arrival order

PriorityQueue quick introduction

- A PriorityQueue returns the smallest Integer first by default
- Adding values places them according to natural ordering
- Poll removes the current smallest; peek reads without removing
- Use when smallest or highest priority must be handled first

```
import java.util.*;
PriorityQueue pq = new
PriorityQueue();
pq.add(Integer.valueOf(30));
pq.add(Integer.valueOf(10));
pq.add(Integer.valueOf(20));
System.out.println(pq.peek());
System.out.println(pq.poll());
System.out.println(pq.poll());
```

Wrap up and connections

- Lists manage ordered data with positions and allow duplicates
- Sets enforce uniqueness and can preserve insertion order
- Maps connect keys to values with clear overwrite rules
- Queues handle items in arrival order or by simple priority
- Next up: Generics & Lambda Expressions!

That’s for today!
Any questions?