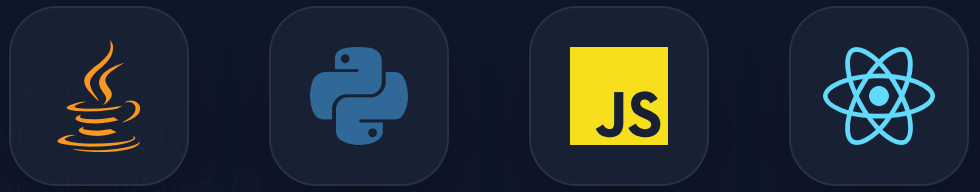

Standard Language APIs

A Comprehensive Guide for Java, Python & JavaScript Developers

Data Structures | Iteration | Sorting | Generics

Advanced Programming tutorial by Dr. Nabajyoti Medhi, PhD, CSE

Table of Contents

SECTION 01

Core Data Structures Overview

How each language models lists, sets, maps, and queues for efficient data organization.

Java

Collections Framework

List, Set, Map, Queue

Python

Built-in Containers

list, tuple, dict, set

JavaScript

Standard Objects

Array, Object, Map, Set

JAVA

Collections Framework

A unified architecture for representing and manipulating collections, allowing them to be manipulated independently of implementation details.

- **List** Ordered sequence. Allows duplicates.
  (e.g., ArrayList, LinkedList)
- **Set** Collection of unique elements.
  (e.g., HashSet, TreeSet)
- **Map** Key-value pairs. Keys are unique.
  (e.g., HashMap, TreeMap)
- **Queue** Elements held for processing (FIFO).
  (e.g., ArrayDeque, PriorityQueue)

JavaCollectionsOverview.java

```java
import java.util.*;

public class JavaCollectionsOverview {
public static void main(String[] args) {
// List: Ordered, allows duplicates
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");

 // Set: Unique elements only
 Set<Integer> uniqueIds = new HashSet<>();
 uniqueIds.add(10);
 uniqueIds.add(20);

 // Map: Key-Value pairs
 Map<Integer, String> idToName = new HashMap<>();
 idToName.put(1, "Admin");
 idToName.put(2, "User");

 // Queue: FIFO ordering
```

PYTHON

Built-in Containers

Python provides versatile, high-level data structures directly in the language, optimized for ease of use and flexibility.

- **list** Mutable, ordered sequence. Dynamic resizing. (equivalent to ArrayList)
- **tuple** Immutable, ordered sequence. Fixed size. (often used for records)
- **dict** Key-value mapping. Ordered by insertion (3.7+). (hash table implementation)
- **set** Unordered collection of unique elements. (mathematical set operations)

python_containers.py

```python
# List: Ordered, mutable, mixed types
names = ["Alice", "Bob", "Carol"]

# Set: Unique elements (curly braces)
unique_ids = {10, 20, 10} # Result: {10, 20}

# Dict: Key-Value pairs
id_to_name = {1: "Admin", 2: "User"}

 # List as a Queue
 task_queue = ["build", "test"]
 task_queue.append("deploy")
 next_task = task_queue.pop(0)

 # Tuple: Immutable sequence
 coords = (10.5, 20.1)
```

JS JAVASCRIPT / REACT

Core Structures

Modern JavaScript provides robust data structures essential for React state management and efficient data handling.

- **Array** Ordered collection. Mutable (but often treated as immutable in React).
- **Object** Key-value map with string or symbol keys. The standard for simple records.
- **Map** True key-value pairs. Keys can be any type (objects, numbers, etc).
- **Set** Collection of unique values. Useful for removing duplicates.

JSDataStructures.js

```
// Array: Ordered list
const names = ["Alice", "Bob", "Carol"];

// Object: Key-value (string keys)
const idToName = { 1: "Admin", 2: "User" };

// Map: Advanced key-value (any key type)
const idToNameMap = new Map([
[1, "Admin"],
 [2, "User"]
 ]);

 // Set: Unique values
 const uniqueIds = new Set([10, 20, 10]);
 // Result: {10, 20}
```

SECTION 02

Lists / Sequences

Master working with ordered collections: insert, update, remove, slice, and sort operations.

Java

List API

ArrayList, LinkedList, methods

Python

List Operations

Append, pop, slice, sort

JavaScript

Arrays in React

Immutable state updates

JAVA

List API Basics

The List interface provides a powerful way to manage ordered collections with precise control over where elements are inserted.

- **Add/Insert** Append to end or insert at specific index.
- **Access** Get or update elements by integer index.
- **Remove** Delete by index or by object value.
- **Sort** Use `Collections.sort()` or `List.sort()`.

JavaListDemo.java

```java
import java.util.*;

public class JavaListDemo {
public static void main(String[] args) {
List<String> names = new ArrayList<>();

// Add elements (append & insert)
names.add("Carol");
names.add(0, "Alice"); // Insert at index 0
 names.add("Bob");

 // Access and Modify
 System.out.println(names.get(1)); // Output: Carol
 names.set(1, "Charlie"); // Update index 1

 // Removal
 names.remove("Bob"); // Remove by value

 // Sorting
 Collections.sort(names); // Alphabetical order
```

PYTHON

Built-in List API

Python lists are mutable, dynamic arrays that support flexible operations for adding, removing, and manipulating elements.

- **Append & Insert** Add elements to the end or at a specific index.
  (O(1) amortized vs O(n))
- **Remove & Pop** Remove by value or index.
  (pop() returns the element)
- **Sort & Reverse** Modify the list in-place for efficiency.
  (Timsort algorithm, O(n log n))
- **Slicing** Powerful syntax to extract sub-lists.
  (e.g., names[0:2])

list_operations.py

```python
names = ["Carol"]

# Adding elements
names.insert(0, "Alice") # At index 0
names.append("Bob") # At the end

print(names[1]) # Output: Carol

# Modifying and Removing
 names[1] = "Charlie" # Update value
 names.remove("Bob") # Remove by value

 # Sorting (In-place)
 names.sort() # Alphabetical

 # Slicing & Popping
 subset = names[0:2] # Get first 2 items
 last = names.pop() # Remove last item
```

REACT / JAVASCRIPT

Immutable Array Patterns

In React, state must be treated as immutable. Never modify arrays directly; create copies to trigger re-renders.

useState Hook Manages array state.
(Always use the setter function)

Spread Syntax Create copies easily.
(e.g., `[...prev, newItem]` to add)

Safe Sorting Copy first, then sort.
(e.g., `[...items].sort()`)

NamesList.jsx

```jsx
import { useState } from "react";

function NamesList() {
const [names, setNames] = useState(["Carol"]);

const addName = () => {
// ✅ Correct: Create new array
setNames(prev => ["Alice", ...prev, "Bob"]);
};

 // ✅ Correct: Copy then sort (don't mutate state)
 const sortedNames = [...names].sort();

 return (
 <div>
 <button onClick={addName}>Add</button>
 <ul>
 {sortedNames.map(name => (
 <li key={name}>{name}</li>
 ))}
```

SECTION 03

Sets and Uniqueness

Mechanisms for ensuring unique elements, performing efficient membership tests, and managing collections without duplicates.

Java

Set Interface

HashSet, LinkedHashSet, TreeSet

Python

Native Sets

set (mutable), frozenset

JavaScript

Set Object

Set, WeakSet for objects

Sets & Uniqueness

Comparing how each language handles unique collections, membership tests, and element removal.

JAVA

SetDemo.java

INIT & ADD

```java
Set<String> tags = new HashSet<>();
tags.add("react");
tags.add("java");
tags.add("react"); // ignored
```

CHECK EXISTS

```java
boolean hasJava =
  tags.contains("java");
```

REMOVE

```java
tags.remove("react");
```

PYTHON

set_demo.py

INIT & ADD

```python
tags = {"react", "java"}
tags.add("react") # ignored
tags.add("python")
```

CHECK EXISTS

```python
print("java" in tags)
```

REMOVE

```python
tags.remove("react")
# or discard() to avoid error
```

JAVASCRIPT

app.js

INIT & ADD

```javascript
const tags = new Set(["react"]);
tags.add("java");
tags.add("react"); // ignored
```

CHECK EXISTS

```javascript
console.log(
  tags.has("java")
);
```

REMOVE

```javascript
tags.delete("react");
```

SECTION 04

Maps / Dictionaries
Key-Value Stores

Explore key–value storage patterns, including efficient lookups, updates, deletion, and iteration across languages.

Java

Map Interface

HashMap, TreeMap, LinkedHashMap

Python

Dictionary (dict)

Hash table based mapping

JavaScript

Object & Map

Object literals, Map class

Advanced Programming tutorial by Dr. Nabajyoti Medhi, PhD, CSE

Maps / Dictionaries

Comparing key–value storage patterns: adding entries, retrieving values, and deletion.

JAVA

MapDemo.java

PUT / INSERT

```java
Map<String, Integer> map = new HashMap<>();
map.put("Alice", 90);
map.put("Bob", 80);
```

GET (W/ DEFAULT)

```java
int score =
  map.getOrDefault("Alice", 0);
```

REMOVE

```java
map.remove("Bob");
// Returns removed value or null
```

PYTHON

dict_demo.py

SET / UPDATE

```python
scores = {"Alice": 90, "Bob": 80}
scores["Alice"] = 95 # overwrite
```

GET (W/ DEFAULT)

```python
score =
  scores.get("Alice", 0)
```

REMOVE

```python
del scores["Bob"]
# or scores.pop("Bob")
```

JAVASCRIPT

object_demo.js

OBJECT PROPERTY

```javascript
const scores = { Alice: 90 };
scores.Alice = 95;
// Map: map.set("Alice", 95)
```

GET / ACCESS

```javascript
const score =
  scores.Alice ?? 0;
```

REMOVE

```javascript
delete scores.Bob;
// Map: map.delete("Bob")
```

SECTION 05

Iteration Patterns

Idiomatic approaches to iterating over sequences and mappings in each language.

Java

Strongly Typed

Enhanced for-loop, Iterator, Streams

Python

Readable Loops

for..in, enumerate(), comprehensions

JavaScript

Functional

map(), for..of, React rendering

Iteration Patterns

Idiomatic ways to loop over sequences and collections in each language.

JAVA

Iteration.java

ENHANCED FOR

```java
for (String name : names) {
  System.out.println(name);
}
```

JAVA 8+ FOREACH

```java
names.forEach(
  System.out::println
);

// Using Iterator (Legacy)
Iterator<String> it =
  names.iterator();
```

PYTHON

iteration.py

FOR LOOP

```python
for name in names:
  print(name)
```

ENUMERATE

```python
# Get index and value
for i, name in enumerate(names):
  print(i, name)
```

DICT ITEMS

```python
for k, v in scores.items():
  print(f"{k}: {v}")
```

JAVASCRIPT

Component.jsx

BASIC ITERATION

```javascript
names.forEach((n, i) => {
  console.log(i, n);
});
```

REACT RENDER

```jsx
// Map to Elements
<ul>
  {names.map(name => (
    <li key={name}>
      {name}
    </li>
  ))}
</ul>
```

SECTION 06

Sorting APIs

Understanding in-place mutation versus sorted copies, custom comparators, and stability across languages.

Java

Collections.sort

Comparator & List.sort()

Python

sort() vs sorted()

key=lambda & stability

JavaScript

Array.prototype.sort

Comparison functions

Sorting APIs

Comparing in-place sorting, sorted copies, and custom comparison logic.

JAVA

SortDemo.java

IN-PLACE

```java
List<Integer> nums = ...;
Collections.sort(nums);
```

CUSTOM COMPARATOR

```java
List<String> names = ...;
names.sort(
  Comparator.comparingInt(
    String::length
  )
);
// Uses Timsort (stable)
```

PYTHON

sort_demo.py

IN-PLACE

```python
nums = [5, 1, 4, 2]
nums.sort()
```

SORTED COPY

```python
sorted_nums = sorted(nums)
```

KEY FUNCTION

```python
users = [{"age": 30}, ...]
users.sort(
  key=lambda u: u["age"]
)
# Also Timsort (stable)
```

JAVASCRIPT

app.js

IN-PLACE

```javascript
const nums = [5, 1, 4, 2];
nums.sort((a, b) => a - b);
// default is string sort!
```

COPY & SORT (REACT)

```javascript
const sortedUsers =
  [...users].sort(
    (a, b) => a.age - b.age
  );
// Stability varies by engine
```

SECTION 07

Generics & Type Parameters

Enabling compile-time type safety and creating reusable, abstract components across languages.

Java

Type Erasure

Compile-time checks: <T>

Python

Type Hints

typing.Generic[T]

TypeScript

Structural Types

interface Props<T>

JAVA

Generics & Type Safety

Generics add stability to your code by making more of your bugs detectable at compile time.

Type Safety Catch type errors early. `List<String>` prevents inserting unrelated objects.

Generic Classes Define classes that operate on any type `T` specified by the client code.

Generic Methods Write a single method declaration that can be called with arguments of different types.

GenericUtils.java

```java
import java.util.*;

public class GenericUtils {

// 1. Generic Class: Reusable container
static class Box<T> {
private T value;
public Box(T value) { this.value = value; }
public T get() { return value; }
 }

 // 2. Generic Method: Works with any List type
 public static <T> void printAll(List<T> list) {
 for (T item : list) {
 System.out.println(item);
 }
 }

 public static void main(String[] args) {
 // 3. Type-Safe Usage
 List<String> names = new ArrayList<>();
 names.add("Alice");
```

Generics & Type Safety

Bringing compile-time safety to dynamic languages using type hints and generics.

Python

TYPING MODULE

generic_box.py

Definition

```python
from typing import TypeVar, Generic
T = TypeVar("T")

class Box(Generic[T]):
    def __init__(self, value: T):
        self.value = value

    def get(self) -> T:
        return self.value
```

Usage

```python
# Type checker ensures safety
str_box: Box[str] = Box("hello")
val: str = str_box.get()
```

TypeScript

GENERICS

GenericBox.ts

Definition

```typescript
class Box<T> {
  private value: T;

  constructor(value: T) {
    this.value = value;
  }

  get(): T {
    return this.value;
  }
}
```

Usage

```typescript
// Compile-time safety
const strBox = new Box<string>("hello");
const val: string = strBox.get();
```

SECTION 08

Higher-Order Iteration

Transform, filter, and reduce collections using functional patterns and declarative streams.

Java

Streams API

stream(), map(), collect()

Python

Comprehensions

[x for x in list if cond]

JavaScript

Array Methods

map, filter, reduce

Advanced Programming tutorial by Dr. Nabajyoti Medhi, PhD, CSE

Higher-Order Iteration

map, filter, and reduce: transforming and aggregating collections functionally.

JAVA

StreamDemo.java

```java
List<Integer> nums = List.of(1,2,3,4,5);

// MAP & FILTER
List<Integer> res = nums.stream()
  .map(n -> n * n)
  .filter(n -> n % 2 == 0)
  .collect(Collectors.toList());

// REDUCE
int sum = nums.stream()
  .reduce(0, Integer::sum);
```

PYTHON

comprehensions.py

```python
nums = [1, 2, 3, 4, 5]

# MAP (COMPREHENSION)
squares = [n * n for n in nums]

# FILTER (COMPREHENSION)
evens = [n for n in squares
         if n % 2 == 0]
# REDUCE (SUM)
total = sum(nums)
```

JAVASCRIPT

app.js

```javascript
const nums = [1, 2, 3, 4, 5];

// MAP
const squares =
  nums.map(n => n * n);
// FILTER
const evens =
  squares.filter(n => n % 2 === 0);
// REDUCE
const total =
  nums.reduce((acc, n) => acc + n, 0);
```

Cross-Language API Comparison

A quick reference guide to equivalent data structures and operations across ecosystems.

FEATURE / CONCEPT	JAVA	PYTHON	JAVASCRIPT
Growable List	`ArrayList<T>`	`list`	`Array`
Key-Value Map	`HashMap<K,V>`	`dict`	`Object` / `Map`
Unique Set	`HashSet<T>`	`set`	`Set`
Sort (In-Place)	`Collections.sort()` *or List.sort()*	`list.sort()`	`arr.sort()`
Sorted Copy	`stream().sorted()` *..collect()*	`sorted(list)`	`[...arr].sort()`
Generics	`<T>`	`TypeVar, Generic`	`<T>`

Summary & Key Takeaways

Essential concepts for mastering cross-language development.

Core Structures Align

Lists, Sets, and Maps exist conceptually in all three ecosystems. The names differ (ArrayList vs list vs Array), but the underlying logic remains consistent.

Idiomatic Patterns

Use the language's strength: Stream API & enhanced loops for Java, list comprehensions for Python, and map/filter/reduce for JavaScript.

Mutability Awareness

Understand when operations modify in-place (Python `sort()`) vs creating copies (Java Streams, JS spread syntax). Crucial for React state.

Type Safety Evolution

Generics provide compile-time safety. Java has them built-in, Python uses Type Hints, and JavaScript relies on TypeScript for similar guarantees.

Practice & Application

The best way to master these APIs is to translate patterns across languages. Try rewriting a data-heavy Java utility in Python or a React component logic in Java to deepen your understanding.