Java Collections Framework

A Comprehensive Guide

Complete Learning Guide

October 9, 2025

Contents

1 Introduction to Collections Framework

1.1 What is Collections Framework?

The Java Collections Framework (JCF) is a unified architecture for representing and manipulating collections of objects. It provides:

- **Interfaces:** Abstract data types representing collections
- **Implementations:** Concrete implementations of collection interfaces
- **Algorithms:** Methods for performing useful computations on collections

1.2 Benefits of Collections Framework

1. **Reduces programming effort:** Provides high-performance data structures
2. **Increases program speed and quality:** Highly optimized implementations
3. **Allows interoperability:** Standard interfaces enable exchange
4. **Reduces effort to learn APIs:** Consistent design patterns
5. **Reduces design effort:** Reusable data structures
6. **Fosters software reuse:** Standard interfaces and implementations

1.3 Collections Framework Hierarchy

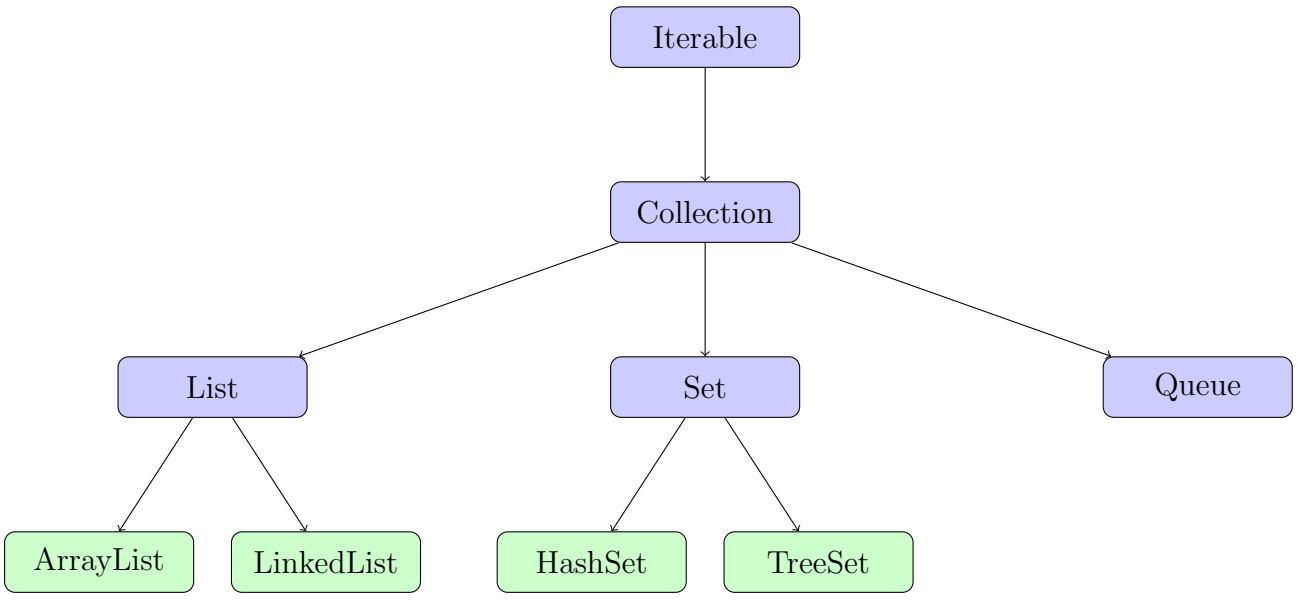

Figure 1: Collections Framework Hierarchy (Partial)

1.4 Map Hierarchy

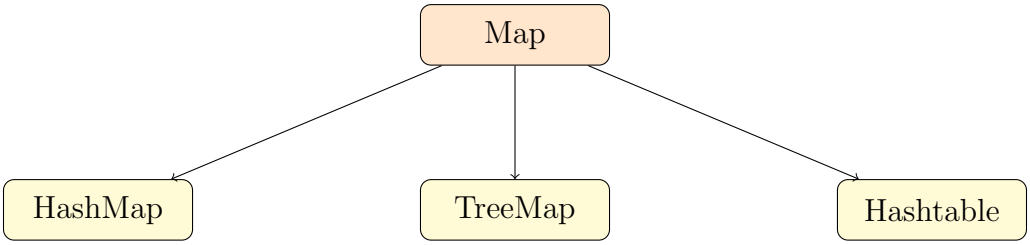

Figure 2: Map Interface Hierarchy

2 Core Interfaces

2.1 Collection Interface

The root interface in the collection hierarchy. It defines basic operations all collections support.

```
// Collection is the root interface
Collection<Integer> nums = new ArrayList<Integer>();

// Basic operations
nums.add(6);          // Add element
nums.remove(6);       // Remove element
nums.size();          // Get size
nums.isEmpty();       // Check if empty
nums.clear();         // Remove all elements
nums.contains(6);     // Check if contains element
```

Listing 1: Collection Interface Example

Key Methods:

Method	Description
boolean add(E e)	Adds element to collection
boolean remove(Object o)	Removes element from collection
int size()	Returns number of elements
boolean isEmpty()	Returns true if empty
void clear()	Removes all elements
boolean contains(Object o)	Checks if element exists
Iterator¡E¿ iterator()	Returns iterator over elements

Table 1: Collection Interface Methods

2.2 List Interface

An ordered collection (sequence). Lists allow:

- Positional access
- Duplicate elements
- Index-based operations

```java
List<Integer> nums2 = new ArrayList<Integer>();

nums2.add(4);            // Add at end
nums2.add(5);
nums2.add(1, 10);        // Insert at index 1
nums2.get(2);            // Access by index
nums2.set(0, 100);       // Replace at index
nums2.remove(1);         // Remove by index
```

Listing 2: List Interface Example

2.2.1 ArrayList vs LinkedList

Operation	**ArrayList**	**LinkedList**
Random Access	O(1) - Fast	O(n) - Slow
Insertion at end	O(1) amortized	O(1)
Insertion at middle	O(n)	O(n)
Deletion	O(n)	O(1) if node known
Memory	Contiguous	Non-contiguous
Best for	Random access	Frequent insertion/deletion

Table 2: ArrayList vs LinkedList Comparison

2.3 Set Interface

A collection that contains no duplicate elements.

```
// HashSet - unordered, O(1) operations
Set<Integer> nums3 = new HashSet<Integer>();
nums3.add(5);
nums3.add(3);
nums3.add(3);  // Duplicate - ignored
// Output: [3, 5] or [5, 3] (unordered)

// TreeSet - sorted, O(log n) operations
Set<Integer> nums4 = new TreeSet<Integer>();
nums4.add(15);
nums4.add(3);
nums4.add(9);
// Output: [3, 9, 15] (sorted)
```

Listing 3: Set Interface Example

2.3.1 Set Implementations Comparison

Feature	**HashSet**	**TreeSet**	**LinkedHashSet**
Order	No order	Sorted order	Insertion order
Performance	O(1)	O(log n)	O(1)
Null elements	One null allowed	No null	One null allowed
Implementation	Hash table	Red-Black tree	Hash table + Linked list
Best for	Fast operations	Sorted data	Maintain order

Table 3: Set Implementations Comparison

2.4 Map Interface

Maps key-value pairs. Each key maps to at most one value.

```
Map<String, Integer> students = new HashMap<>();

// Add key-value pairs
students.put("Navin", 56);
students.put("Rahul", 78);
students.put("Priya", 92);

// Access value by key
int marks = students.get("Navin");  // Returns 56

// Check for key/value
boolean hasKey = students.containsKey("Navin");
boolean hasValue = students.containsValue(78);

```

```java
// Update value
students.put("Navin", 65);  // Updates existing key

// Remove entry
students.remove("Rahul");

// Iterate over entries
for(Map.Entry<String, Integer> entry : students.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

Listing 4: Map Interface Example

2.4.1 HashMap vs Hashtable vs TreeMap

Feature	HashMap	Hashtable	TreeMap
Thread-safe	No	Yes (synchronized)	No
Null keys	One null key	No null keys	No null keys
Null values	Multiple null values	No null values	Null values allowed
Order	No order	No order	Sorted by keys
Performance	O(1)	O(1)	O(log n)
Legacy	No	Yes (legacy)	No
Best for	General use	Thread-safe	Sorted keys

Table 4: Map Implementations Comparison

3 Iterator Interface

Iterator provides a way to traverse collections element by element.

```
Collection<Integer> nums = new ArrayList<>();
nums.add(6);
nums.add(7);
nums.add(8);

// Get iterator
Iterator<Integer> values = nums.iterator();

// Traverse using iterator
while(values.hasNext()) {
    Integer value = values.next();
    System.out.println(value);
}

// Can also remove elements while iterating
Iterator<Integer> iter = nums.iterator();
while(iter.hasNext()) {
    Integer val = iter.next();
    if(val == 7) {
        iter.remove();  // Safe removal
    }
}
```

Listing 5: Iterator Example

Iterator Methods:

Method	Description
boolean hasNext()	Returns true if more elements exist
E next()	Returns next element
void remove()	Removes last element returned by next()

Table 5: Iterator Methods

3.1 Enhanced For Loop vs Iterator

```java
// Enhanced for-loop (for-each)
for(Integer n : nums) {
    System.out.println(n);
    // Cannot remove elements - throws
       ConcurrentModificationException
}

// Iterator - allows removal
Iterator<Integer> iter = nums.iterator();
while(iter.hasNext()) {
    Integer n = iter.next();
    if(condition) {
        iter.remove();  // Safe removal
    }
}
```

Listing 6: For-Each vs Iterator

4 Generics

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods.

4.1 Why Generics?

1. **Type Safety:** Compile-time type checking
2. **No Casting:** Eliminates explicit casting
3. **Code Reusability:** Write once, use with any type

```
// WITHOUT Generics - Old way
List list = new ArrayList();
list.add("Hello");
list.add(10);  // No compile error, but runtime error possible
String s = (String) list.get(0);  // Explicit cast needed

// WITH Generics - Type safe
List<String> list2 = new ArrayList<String>();
list2.add("Hello");
// list2.add(10);  // Compile-time error!
String s2 = list2.get(0);  // No cast needed
```

Listing 7: Without vs With Generics

4.2 Generic Syntax

Symbol	Meaning
E	Element (used in collections)
K	Key (used in maps)
V	Value (used in maps)
N	Number
T	Type

Table 6: Generic Type Parameter Naming Conventions

5 Wrapper Classes

Wrapper classes convert primitive types into objects.

5.1 Why Wrapper Classes?

- Collections work only with objects, not primitives
- Provide utility methods for conversion and parsing
- Enable nullability (primitives cannot be null)

5.2 Primitive to Wrapper Mapping

Primitive	**Wrapper Class**	**Example**
byte	Byte	Byte b = 127;
short	Short	Short s = 32000;
int	Integer	Integer i = 100;
long	Long	Long l = 1000L;
float	Float	Float f = 3.14f;
double	Double	Double d = 3.14159;
char	Character	Character c = 'A';
boolean	Boolean	Boolean b = true;

Table 7: Primitive to Wrapper Class Mapping

5.3 Auto-boxing and Auto-unboxing

```
// Auto-boxing: primitive -> wrapper
int primitive = 10;
Integer wrapper = primitive;  // Automatically converted
// Internally: Integer wrapper = Integer.valueOf(primitive);

// Auto-unboxing: wrapper -> primitive
Integer wrapperObj = 20;
int primitiveValue = wrapperObj;  // Automatically converted
// Internally: int primitiveValue = wrapperObj.intValue();

// Collections use auto-boxing
List<Integer> list = new ArrayList<>();
list.add(5);  // Auto-boxing: 5 -> Integer.valueOf(5)
int value = list.get(0);  // Auto-unboxing: Integer -> int
```

Listing 8: Auto-boxing and Auto-unboxing

6 Collections Utility Class

The `Collections` class provides static methods for operating on collections.

```
List<Integer> list = new ArrayList<>();
list.add(45);
list.add(12);
list.add(78);
list.add(23);

// Sorting
Collections.sort(list);  // [12, 23, 45, 78]

// Reverse
Collections.reverse(list);  // [78, 45, 23, 12]

// Shuffle
Collections.shuffle(list);  // Random order

// Binary search (list must be sorted)
Collections.sort(list);
int index = Collections.binarySearch(list, 45);

// Min and Max
int min = Collections.min(list);
int max = Collections.max(list);

// Frequency
int count = Collections.frequency(list, 45);

// Fill
Collections.fill(list, 0);  // Replace all with 0
```

Listing 9: Collections Utility Methods

6.1 Common Collections Methods

Method	Description
sort(List)	Sorts list in natural order
reverse(List)	Reverses order of elements
shuffle(List)	Randomly permutes elements
binarySearch(List, Object)	Searches for element (list must be sorted)
min(Collection)	Returns minimum element
max(Collection)	Returns maximum element
frequency(Collection, Object)	Counts occurrences of element
fill(List, Object)	Replaces all elements with specified value
copy(List dest, List src)	Copies all elements from one list to another

Table 8: Common Collections Utility Methods

7 Comparator Interface

The Comparator interface defines custom comparison logic for sorting objects.

7.1 Comparator vs Comparable

Feature	Comparable	Comparator
Package	java.lang	java.util
Method	compareTo(Object)	compare(Object, Object)
Usage	Natural ordering	Custom ordering
Modification	Modify class itself	Separate class
Example	String, Integer	Custom comparators

Table 9: Comparable vs Comparator

7.2 Comparator Implementation

```
List<Integer> nums = new ArrayList<>();
nums.add(43);
nums.add(31);
nums.add(72);
nums.add(29);

// Sort by last digit using anonymous class
Comparator<Integer> lastDigitComparator = new Comparator<Integer
   >() {
    @Override
    public int compare(Integer i, Integer j) {
        // Positive: i > j (i comes after j)
        // Negative: i < j (i comes before j)
        // Zero: i == j (equal)
        return Integer.compare(i % 10, j % 10);
    }
};

Collections.sort(nums, lastDigitComparator);
// Result: [31, 72, 43, 29] (sorted by last digit)

// Lambda expression (Java 8+)
Collections.sort(nums, (i, j) -> Integer.compare(i % 10, j % 10))
   ;

// Method reference (even more concise)
Collections.sort(nums, Comparator.comparingInt(i -> i % 10));
```

Listing 10: Comparator Examples

7.3 Comparator Return Values

Return Value	Meaning	Result
Positive (¿ 0)	First ¿ Second	First comes after second
Negative (¡ 0)	First ¡ Second	First comes before second
Zero (= 0)	First == Second	Equal (order unchanged)

Table 10: Comparator Return Values

7.4 Sorting Custom Objects

```
class Student {
    int age;
    String name;

    public Student(int age, String name) {
        this.age = age;
        this.name = name;
    }
}

List<Student> students = new ArrayList<>();
students.add(new Student(20, "Alice"));
students.add(new Student(22, "Bob"));
students.add(new Student(19, "Charlie"));

// Sort by age - Anonymous inner class
Collections.sort(students, new Comparator<Student>() {
    @Override
    public int compare(Student s1, Student s2) {
        return Integer.compare(s1.age, s2.age);
    }
});

// Sort by name - Lambda expression
Collections.sort(students, (s1, s2) -> s1.name.compareTo(s2.name)
   );

// Multiple criteria sorting
Collections.sort(students, new Comparator<Student>() {
    @Override
    public int compare(Student s1, Student s2) {
        // Primary: sort by age
        int ageCompare = Integer.compare(s1.age, s2.age);
        if (ageCompare != 0) {
            return ageCompare;
        }
        // Secondary: if ages equal, sort by name
        return s1.name.compareTo(s2.name);
    }
});
```

Listing 11: Sorting Student Objects

8 Anonymous Inner Classes

An anonymous class is a class without a name, defined and instantiated in a single expression.

8.1 What is an Anonymous Inner Class?

An anonymous inner class is:

- A class without a name
- Defined and instantiated at the same time
- Used for one-time implementations
- Can extend a class or implement an interface

8.2 Syntax and Examples

```
// Syntax for implementing an interface
InterfaceName variable = new InterfaceName() {
    // Override methods
    @Override
    public void method() {
        // Implementation
    }
};

// Example 1: Comparator
Comparator<Integer> comp = new Comparator<Integer>() {
    @Override
    public int compare(Integer i, Integer j) {
        return Integer.compare(i, j);
    }
};

// Example 2: Runnable
Runnable task = new Runnable() {
    @Override
    public void run() {
        System.out.println("Task running");
    }
};
task.run();

// Example 3: ActionListener (GUI)
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked");
    }
```

```
33 });
```

Listing 12: Anonymous Inner Class Syntax

8.3 Anonymous Class vs Lambda Expression

```
// Anonymous Inner Class (verbose)
Comparator<Integer> comp1 = new Comparator<Integer>() {
    @Override
    public int compare(Integer i, Integer j) {
        return i - j;
    }
};

// Lambda Expression (concise) - Java 8+
Comparator<Integer> comp2 = (i, j) -> i - j;

// Both do the same thing, but lambda is more concise
// Lambda works only with functional interfaces (single abstract
   method)
```

Listing 13: Anonymous Class vs Lambda

Feature	Anonymous Class	Lambda Expression
Verbosity	More verbose	Very concise
'this' keyword	Refers to anonymous class	Refers to enclosing class
Variables	Can have instance variables	Cannot have instance variables
Interfaces	Can implement multiple interfaces	Only functional interfaces
Use case	Complex implementations	Simple, single-method implementations

Table 11: Anonymous Class vs Lambda Expression

9 Interfaces in Java

An interface is a reference type that contains only abstract methods and constants.

9.1 What is an Interface?

- A contract that classes must follow
- Contains method signatures without implementations
- Supports multiple inheritance
- Cannot be instantiated directly

```
// Defining an interface
interface Animal {
    void eat();    // Abstract method (no body)
    void sleep();  // Abstract method
}
```

```java

// Implementing interface
class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("Dog is eating");
    }

    @Override
    public void sleep() {
        System.out.println("Dog is sleeping");
    }
}

// Using the interface
Animal animal = new Dog();
animal.eat();
animal.sleep();
```

Listing 14: Interface Example

9.2 Functional Interfaces

A functional interface has exactly one abstract method. Used with lambda expressions.

```java
// Comparator - functional interface
@FunctionalInterface
interface Comparator<T> {
    int compare(T o1, T o2);  // Single abstract method
}

// Runnable - functional interface
@FunctionalInterface
interface Runnable {
    void run();  // Single abstract method
}

// Custom functional interface
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}

// Using with lambda
Calculator add = (a, b) -> a + b;
Calculator multiply = (a, b) -> a * b;

System.out.println(add.calculate(5, 3));        // 8
System.out.println(multiply.calculate(5, 3));  // 15
```

Listing 15: Functional Interface Examples

10 Performance Comparison

10.1 Time Complexity Table

Operation	ArrayList	LinkedList	HashSet	TreeSet
Add	O(1) amortized	O(1)	O(1)	O(log n)
Remove	O(n)	O(1)*	O(1)	O(log n)
Get/Access	O(1)	O(n)	N/A	N/A
Contains	O(n)	O(n)	O(1)	O(log n)
Iteration	O(n)	O(n)	O(n)	O(n)
*O(1) if node reference is known, otherwise O(n)				

Table 12: Time Complexity Comparison

10.2 Space Complexity

Data Structure	Space Complexity	Extra Space
ArrayList	O(n)	Contiguous array
LinkedList	O(n)	Node pointers (2x space)
HashSet	O(n)	Hash table + buckets
TreeSet	O(n)	Tree nodes + pointers
HashMap	O(n)	Hash table + key-value pairs
TreeMap	O(n)	Tree nodes + key-value pairs

Table 13: Space Complexity Comparison

11 Best Practices

11.1 When to Use Which Collection

Use Case	Best Choice
Fast random access	ArrayList
Frequent insertions/deletions	LinkedList
No duplicates needed	HashSet
Sorted unique elements	TreeSet
Key-value pairs	HashMap
Sorted key-value pairs	TreeMap
Thread-safe operations	ConcurrentHashMap, Vector
Maintain insertion order	LinkedHashSet, LinkedHashMap
FIFO queue	LinkedList, ArrayDeque
Priority queue	PriorityQueue

Table 14: Collection Selection Guide

11.2 Programming Guidelines

1. **Program to interface:** Use `List<>` instead of `ArrayList<>`

```
List<String> list = new ArrayList<>();  // Good
ArrayList<String> list = new ArrayList<>();  // Avoid
```

2. **Use generics:** Always specify type parameters

```
List<Integer> list = new ArrayList<>();  // Good
List list = new ArrayList();  // Avoid (raw type)
```

3. **Initialize with capacity:** For known sizes

```
List<Integer> list = new ArrayList<>(1000);  // Efficient
```

4. **Use appropriate collection:** Choose based on operations

5. **Avoid null checks:** Use `Collections.emptyList()` instead

12 Complete Code Example

```
package Collections;

import java.util.*;

class Student {
    int age;
    String name;

    public Student(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return "Student{name='" + name + "', age=" + age + "}";
    }
}

public class Demo {
    public static void main(String[] args) {

        // 1. COLLECTION INTERFACE
        Collection<Integer> nums = new ArrayList<Integer>();
        nums.add(6);
        nums.add(7);
        nums.add(8);

        for(Integer n : nums) {
            System.out.println(n * 2);
        }

        // 2. LIST INTERFACE
        List<Integer> nums2 = new ArrayList<Integer>();
        nums2.add(4);
        nums2.add(1, 10);  // Insert at index
        System.out.println(nums2.get(0));

        // 3. SET INTERFACE
        Set<Integer> nums3 = new HashSet<Integer>();
        nums3.add(5);
        nums3.add(3);
        nums3.add(3);  // Duplicate ignored

        Set<Integer> nums4 = new TreeSet<Integer>();
        nums4.add(15);
        nums4.add(3);
        nums4.add(9);  // Automatically sorted

```

```
50          // 4. ITERATOR
51          Iterator<Integer> values = nums.iterator();
52          while(values.hasNext()) {
53              System.out.println(values.next());
54          }
55
56          // 5. MAP INTERFACE
57          Map<String, Integer> students = new HashMap<>();
58          students.put("Navin", 56);
59          students.put("Rahul", 78);
60
61          for(Map.Entry<String, Integer> entry : students.entrySet
                ()) {
62              System.out.println(entry.getKey() + ": " + entry.
                    getValue());
63          }
64
65          // 6. COLLECTIONS UTILITY
66          List<Integer> sortList = new ArrayList<>();
67          sortList.add(45);
68          sortList.add(12);
69          sortList.add(78);
70
71          Collections.sort(sortList);
72          System.out.println("Sorted: " + sortList);
73
74          // 7. COMPARATOR INTERFACE
75          Comparator<Integer> lastDigitComp = new Comparator<
                Integer>() {
76              @Override
77              public int compare(Integer i, Integer j) {
78                  return Integer.compare(i % 10, j % 10);
79              }
80          };
81
82          Collections.sort(sortList, lastDigitComp);
83
84          // Lambda version
85          Collections.sort(sortList, (i, j) -> Integer.compare(i %
                10, j % 10));
86
87          // 8. WRAPPER CLASSES
88          int primitive = 10;
89          Integer wrapper = primitive;  // Auto-boxing
90          int back = wrapper;           // Auto-unboxing
91
92          // 9. CUSTOM OBJECTS
93          List<Student> studentList = new ArrayList<>();
94          studentList.add(new Student(20, "Alice"));
95          studentList.add(new Student(22, "Bob"));
96
```

```
97          Collections.sort(studentList, (s1, s2) -> Integer.compare
                (s1.age, s2.age));
98      }
99  }
```

Listing 16: Complete Demo.java Implementation

13 Common Pitfalls and Solutions

13.1 ConcurrentModificationException

```
// WRONG - throws ConcurrentModificationException
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);

for(Integer n : list) {
    if(n == 2) {
        list.remove(n);  // ERROR!
    }
}

// CORRECT - use Iterator
Iterator<Integer> iter = list.iterator();
while(iter.hasNext()) {
    Integer n = iter.next();
    if(n == 2) {
        iter.remove();  // Safe removal
    }
}
```

Listing 17: Concurrent Modification Error

13.2 Null Pointer with Auto-unboxing

```
// WRONG - throws NullPointerException
List<Integer> list = new ArrayList<>();
list.add(null);
int value = list.get(0);  // Auto-unboxing null -> ERROR!

// CORRECT - check for null
Integer wrapper = list.get(0);
if(wrapper != null) {
    int value = wrapper;
}
```

Listing 18: Null Pointer Error

13.3 HashCode and Equals

```
class Student {
    String name;
    int age;

    // Must override both for proper HashSet/HashMap behavior
    @Override
```

```
7       public boolean equals(Object o) {
8           if(this == o) return true;
9           if(o == null || getClass() != o.getClass()) return false;
10          Student student = (Student) o;
11          return age == student.age && name.equals(student.name);
12      }
13
14      @Override
15      public int hashCode() {
16          return Objects.hash(name, age);
17      }
18  }
```

Listing 19: Override hashCode and equals

14 Advanced Topics

14.1 Stream API (Java 8+)

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9,
    10);

// Filter and collect
List<Integer> evens = numbers.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());

// Map and sum
int sum = numbers.stream()
    .map(n -> n * 2)
    .reduce(0, Integer::sum);

// Sorting with streams
List<String> sorted = names.stream()
    .sorted()
    .collect(Collectors.toList());
```

Listing 20: Stream API with Collections

14.2 Concurrent Collections

```
// ConcurrentHashMap - thread-safe, better than Hashtable
Map<String, Integer> concurrentMap = new ConcurrentHashMap<>();

// CopyOnWriteArrayList - thread-safe list
List<String> concurrentList = new CopyOnWriteArrayList<>();

// BlockingQueue - producer-consumer pattern
BlockingQueue<Integer> queue = new LinkedBlockingQueue<>();
```

Listing 21: Thread-Safe Collections

15 Summary

15.1 Key Takeaways

1. **Collections Framework** provides unified architecture for data structures
2. **List**: Ordered, allows duplicates (ArrayList, LinkedList)
3. **Set**: Unordered, no duplicates (HashSet, TreeSet)
4. **Map**: Key-value pairs (HashMap, TreeMap, Hashtable)
5. **Iterator**: Safe way to traverse and remove elements

6. **Generics**: Type safety at compile time
7. **Wrapper Classes**: Enable primitives in collections
8. **Comparator**: Custom sorting logic
9. **Anonymous Classes**: One-time implementations
10. **Collections Utility**: Helper methods for operations

15.2 Decision Tree for Collection Selection

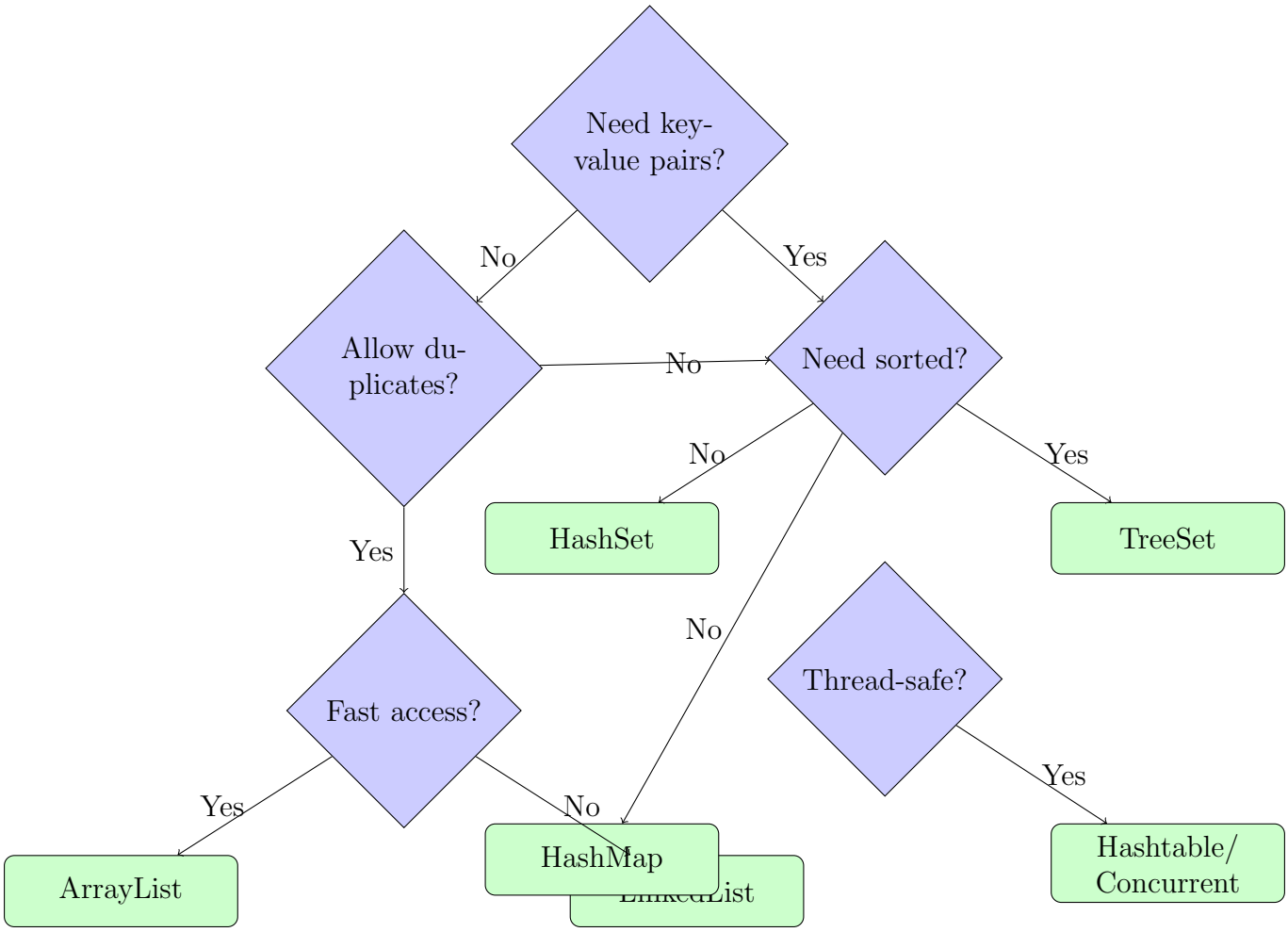

Figure 3: Collection Selection Decision Tree

16 References and Further Reading

- Oracle Java Documentation: `https://docs.oracle.com/javase/tutorial/collections/`
- Effective Java by Joshua Bloch
- Java Collections Framework Official Guide
- Java Generics and Collections by Maurice Naftalin