Collections

JoeDayz | 2024

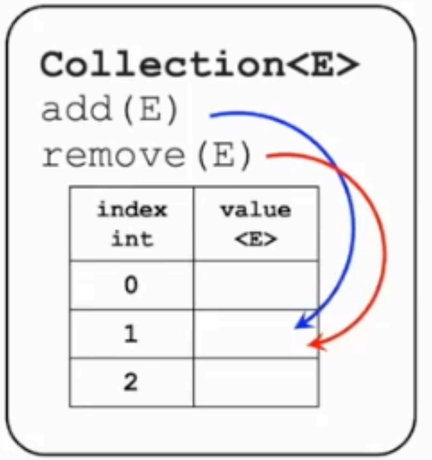
Objectives

After completing this lesson, you should be able to:

- Introduce Java Collection API interfaces and implementation classes
- Use List, Set, Deque, and Map collections
- Iterate through collection content
- Use Collections class
- Access collections concurrently

Introduction to Java Collection API

- Collection API presents a number of classes that manipulate groups of objects (collections).
- Collection classes may provide features such as:
  - Use generics
  - Dynamically expand as appropriate
  - Provide alternative search and index capabilities
  - Validate elements within the collection (for example, check uniqueness)
  - Order elements
  - Provide thread-safe operations (protect internal storage from corruption when it is accessed concurrently from multiple threads)
  - Iterate through the collection
- Collection API defines a number of interfaces describing such capabilities.
- Collection API classes implement these interfaces and can provide different combinations of capabilities.



✣ **Note:** Collection API provides much better flexibility compared to just using arrays.

Java Collection API Interfaces

Java Collection API located in the `java.util` package provides automations and convenience wrappers for various types of collections.

Collection API interfaces:

- `Iterable<T>` iterate through collection.
- `Collection<E>` add and remove elements.
- `SequencedCollection<E>` * describes a collection with a well-defined encounter order of elements.
- `List<E>` describes a list of elements indexed by `int`.
- `Deque<E>` describes a double-ended queue, providing (FIFO) and (LIFO) behaviors.
- `Set<E>` describes a set of unique elements.
- `SequencedSet<E>` * is a set-specific variant of `SequencedCollection`.
- `SortedSet<E>` adds ordering ability.
- `Map<K,V>` interface contains a `Set` of unique keys and a `Collection` of values.
- `SequencedMap<E>` * describes a map with a well-defined encounter order of elements.

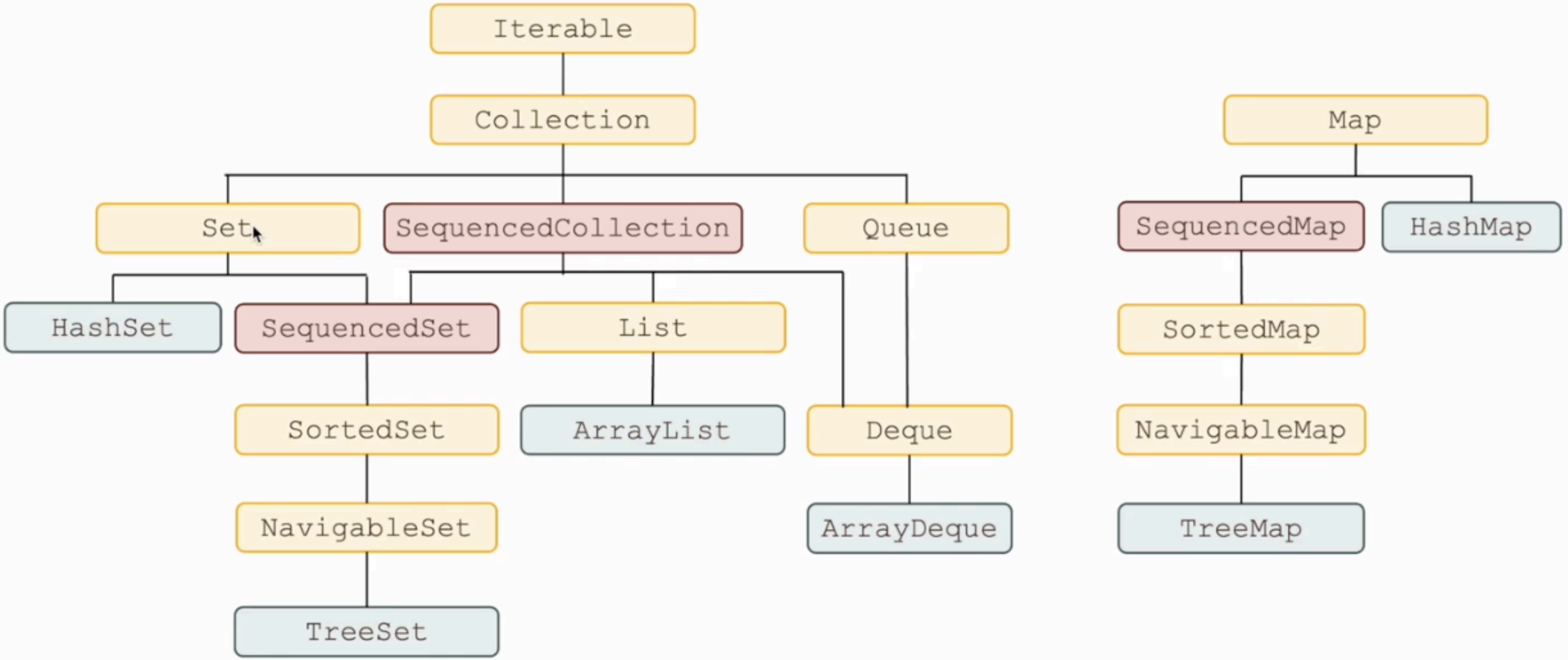
✣ **Note:** SequencedCollection, SequencedSet, and SequencedMap are Java SE 21 new features.

Java Collection API Implementation Classes

Examples of Collection API classes:

- `ArrayList<E>` implements `List` interface.
- `ArrayDeque<E>` implements the `Deque` interface.
- `HashSet<E>` implements the `Set` interface.
- `TreeSet<E>` implements the `NavigableSet` interface.
- `HashMap<K,V>` implements the `Map` interface.
- `TreeMap<K,V>` implements the `NavigableMap` interface.

Java Collection API Interfaces and Implementation Classes



Create List Object

Class `java.util.ArrayList` is an implementation of the `List` interface.

- Instances of `ArrayList` can be created using:
  - No-arg constructor, creating a list of initial capacity of 10 elements
  - Constructor with specific initial capacity
  - Any other `Collection` (for example, `Set`) to populate this list with initial values
- A fixed-sized `List` can be created from the array using var-arg method `Arrays.asList(<T>...)`
- A read-only instance of `List` can be created using a var-arg method `List.of(<T> ...)`
- ArrayList will auto-expand its internal storage, when more elements are added to it.

```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
Set<Product> set1 = new HashSet<>();
set1.add(p1);
set1.add(p2);

List<Product> list1 = new ArrayList<>();
List<Product> list2 = new ArrayList<>(20);
List<Product> list3 = new ArrayList<>(set1);
List<Product> list4 = Arrays.asList(p1,p2);
List<Product> list5 = List.of(p1,p2);
```

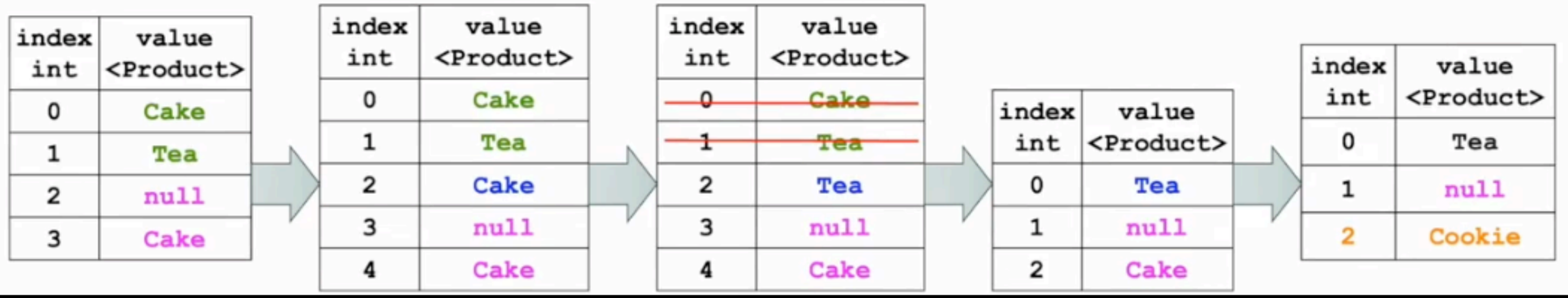
- **Note:** HashSet is explained later.
- **Reminder:** var-arg parameter accepts a number of parameters or an array.

Manage List Contents

`List` represents collection of elements indexed by `int`

- Insert
  - `boolean add(T)`
  - `boolean add(int, T)`
- Update
  - `boolean set(int, T)`
- Delete
  - `boolean remove(int)`
  - `boolean remove(T)`
- Check Existence
  - `boolean contains(T)`
- Find an index
  - `int indexOf(T)`

```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
List<Product> menu = new ArrayList<>();
menu.add(p1); //insert first element
menu.add(p2); //insert next element
menu.add(2, null); //insert null
menu.add(3, p1);   //insert element
menu.add(2, p1);   //insert element
menu.set(2, p2);   //update element
menu.remove(0);    //remove element
menu.remove(p2);   //remove element
boolean hasTea = menu.contains(p2);
int index = menu.indexOf(p1);
menu.get(index).setName("Cookie");
menu.add(4, p2);  // throws exception
```



Create Set Object

Class `java.util.HashSet` is an implementation of the `Set` interface.

- Instances of `HashSet` can be created using:
  - No-arg constructor, creating a set of initial capacity of 16 elements
  - Constructor with specific initial capacity
  - Constructor with specific initial capacity and load factor (default is 0.75)
  - Any other `Collection` (for example, `List`) to populate this set with initial values
- HashSet will auto-expand its internal storage when more elements are added to it.
- A read-only instance of `Set` can be created using a var-arg method `Set.of(<T> ...)`

```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
List<Product> list = new ArrayList<>();
list.add(p1);
list.add(p2);

Set<Product> productSet1 = new HashSet<>();
Set<Product> productSet2 = new HashSet<>(20);
Set<Product> productSet3 = new HashSet<>(20, 0.85);
Set<Product> productSet4 = new HashSet<>(list);
Set<Product> productSet5 = Set.of(p1,p2);
```

- The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.
- List allows duplicate elements while Set does not; when Set is populated based on List, duplicate entries are discarded.
- Reminder: var-arg parameter accepts a number of parameters or an array.

Manage Set Contents

Set represents collection of unique elements.

- Insert `boolean add(T)`
- Delete `boolean remove(T)`
- Check Existence `boolean contains(T)`
- Duplicate element cannot be added to the set:
  - Methods `add` and `remove` verify if the element exists in the set using the `equals` method.
  - Methods `add` and `remove` will return `false` value when attempting to add a duplicate or remove an absent element.


```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
Product p3 = new Food("Cookie");
Set<Product> menu = new HashSet<>();
menu.add(p1);     //insert element
menu.add(p2);     //insert element
menu.add(p2);     //insert nothing
menu.add(p3);     //insert element
menu.remove(p1); //remove element
menu.remove(p1); //remove nothing
boolean hasTea = menu.contains(p2);
```

Create Deque Object

Class `java.util.ArrayDeque` is an implementation of the `Deque` interface.

- Instances of `ArrayDeque` can be created using:
  - No-arg constructor, creating a set of initial capacity of 16 elements
  - Constructor with specific initial capacity
  - Any other `Collection` (for example, `List`) to populate this deque with initial values
- `ArrayDeque` will auto-expand as more elements are added to it.

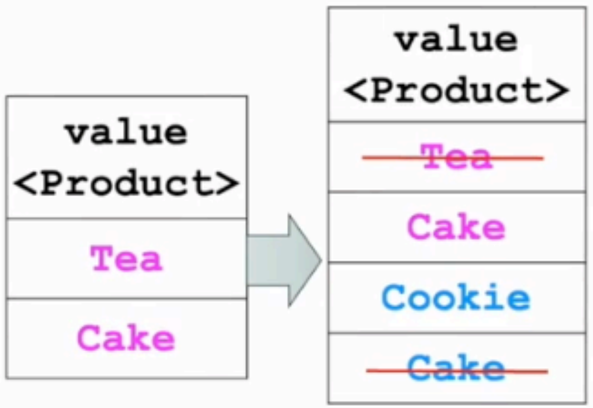
```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
List<Product> list = new ArrayList<>();
list.add(p1);
list.add(p2);

Deque<Product> productDeque1 = new ArrayDeque<>();
Deque<Product> productDeque2 = new ArrayDeque<>(20);
Deque<Product> productDeque3 = new ArrayDeque<>(list);
```

Manage Deque Contents

Deque represents collection that implements FIFO, LIFO behaviors.

- `offerFirst(T)` and `offerLast(T)` insert elements at the head and the tail of the deque.
- `T pollFirst()` and `T pollLast()` get and remove elements at the head and the tail of the deque.
- `T peekFirst()` and `T peekLast()` get elements at the head and the tail of the deque.
- Null values are not allowed in a deque.
- If deque is empty, poll and peek operations return null.




```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
Product p3 = new Drink("Cookie");
Deque<Product> menu = new ArrayDeque<>();
Product nullProduct = menu.pollFirst();
menu.offerFirst(p1);
menu.offerFirst(p2);
Product tea = menu.pollFirst();
Product cake1 = menu.peekFirst();
menu.offerLast(p3);
menu.offerLast(p1);
Product cake2 = menu.pollLast();
Product cookie = menu.peekLast();
menu.offerLast(null);
```

Create HashMap Object

Class `java.util.HashMap` is an implementation of the `Map` interface.

`Map` is a composition of a `Set` of keys and a `Collection` of values.

- Instances of `HashMap` can be created using:
  - No-arg constructor, creating a set of initial capacity of 16 elements
  - Constructor with specific initial capacity
  - Constructor with specific initial capacity and load factor (default is 0.75)
  - Any other `Map` to populate this set with initial values
- HashMap will auto-expand its internal storage, when more elements are added to it.
- A read-only instance of `Map` can be created using
  - Method `of(<key>,<value>, ...)` overloaded for up to 10 map entries
  - A var-arg method `ofEntries (Map.entry<key, value>... entries)`

```
Map<Product, Integer> items1 = new HashMap<>();
Map<Product, Integer> items2 = new HashMap<>(20);
Map<Product, Integer> items3 = new HashMap<>(items1);
Map<Product, Integer> items4 = Map.of(new Food("Cake"), Integer.valueOf(2),
                                      new Drink("Tea"), Integer.valueOf(3));
```

Manage HashMap Contents

`Map` represents collection of values with unique keys.

- `V put(K, V)` : Insert or update (when using an existing key) a key-value pair
- `V remove(K)` : Delete a key-value pair
- `V get(K)` : Return the value for a given key
- `boolean containsKey(K)` : Check existence of a key
- `boolean containsValue(V)` : Check existence of a value

key <Product>	value <Integer>
Cake	2 ⇨ 5
~~Tea~~	~~2~~

```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
Map<Product, Integer> items = new HashMap<>();
items.put(p1,Integer.valueOf(2));
items.put(p2,Integer.valueOf(2));
Integer n1 = items.put(p1,Integer.valueOf(5));
Integer n2 = items.remove(p2);
boolean hasTea = items.containsKey(p2);
boolean hasTwo = items.containsValue(n1);
Integer quantity = items.get(p1);
```