Java 21

Iterando colecciones

2025

Iterate Through Collections

Collections implement the `Iterable` interface, allowing them to be used in a `forEach` loop.

- Any `List`, `Set`, or `Deque` can be directly used within a `forEach` loop.
- A more manual approach is to get an `Iterator` object from collection to step through the content.
- Get `Set` of keys or `List` of values from the `HashMap` to iterate through the content.
- Iterators also allow to remove content from the collection.

```
Map<Product,Integer> items = new HashMap<>();
Set<Product> keys = items.keySet();
Collection<Integer> values = items.values();
for (Product product : keys) {
  Integer quantity = items.get(product);
  // use product and quantity objects
}
for (Integer quantity : values) {
  // use quantity object
}
```

```
List<Product> menu = new ArrayList<>();
menu.add(new Food("Cake"));
menu.add(new Drink("Tea"));
for (Product product : menu) {
  // use product object
}

// less automated alternative:
Iterator<Product> iter = menu.iterator();
while (iter.hasNext()) {
  Product product = iter.next();
  // use product object
  iter.remove();
}
```

Sequenced Collections

Sequenced Collections are variants of the Collection interface:

- Describe a well-defined encounter order of elements.
- Support operations for elements at the start and the end of the collection.
- Add the reverse capability.
- Are not the same as sorted collections!

```
interface SequencedMap<K,V>
          extends Map<K,V> {
  // New methods:
  SequencedMap<K,V> reversed();
  SequencedSet<K> sequencedKeySet();
  SequencedCollection<V> sequencedValues();
  SequencedSet<Entry<K,V>> sequencedEntrySet();
  V putFirst(K, V);
  V putLast(K, V);
  // Methods promoted from NavigableMap interface:
  Entry<K, V> firstEntry();
  Entry<K, V> lastEntry();
  Entry<K, V> pollFirstEntry();
  Entry<K, V> pollLastEntry();
}
```

```
interface SequencedCollection<E>
          extends Collection<E> {
    // New method:
    SequencedCollection<E> reversed();
    // Methods promoted from Deque interface:
    void addFirst(E);
    void addLast(E);
    E getFirst();
    E getLast();
    E removeFirst();
    E removeLast();
}
```

```
interface SequencedSet<E>
          extends SequencedCollection<E>,
                  Set<E> {
  // New method:
  SequencedSet<E> reversed();
}
```

```java
SequencedMap<String, Integer> map = new LinkedHashMap<>();
map.put("A", 1);
map.put("B", 2);
map.putFirst("C", 3);  // Inserta al inicio

System.out.println(map); // {C=3, A=1, B=2}

SequencedMap<String, Integer> reversedMap = map.reversed();
System.out.println(reversedMap); // {B=2, A=1, C=3}
```

```java
SequencedSet<String> set = new LinkedHashSet<>();
set.add("One");
set.add("Two");
set.addFirst("Zero");

System.out.println(set); // [Zero, One, Two]
set.removeLast();        // Elimina "Two"

System.out.println(set); // [Zero, One]
```

Other Collection Behaviors

Some other examples of generic Collection behaviors include:

- Convert collection to an array using the `toArray` method
  - Use array provided if collection fits into it
  - Otherwise create new array with the matching size
- Remove elements from collection based on a condition
  - Implement interface `Predicate<T>` overriding abstract method `boolean test(T);`
  - Use `removeIf(Predicate<T> p)` method to remove all matching elements from the collection

```
import java.util.function.Predicate;
public class LongProductsNames
        implements Predicate<Product> {
  public boolean test(Product product) {
    return product.getName().length() > 3;
  }
}
```

```
List<Product> menu = new ArrayList<>();
menu.add(new Food("Cake"));
menu.add(new Drink("Tea"));
menu.add(new Food("Cookie"));

Product[] array = new Product[2];
array = menu.toArray(array);

menu.removeIf(new LongProductsNames());
```

Note for the example:

- Product array is re-created to accommodate extra elements.
- Method `removeIf` removes all products except tea.

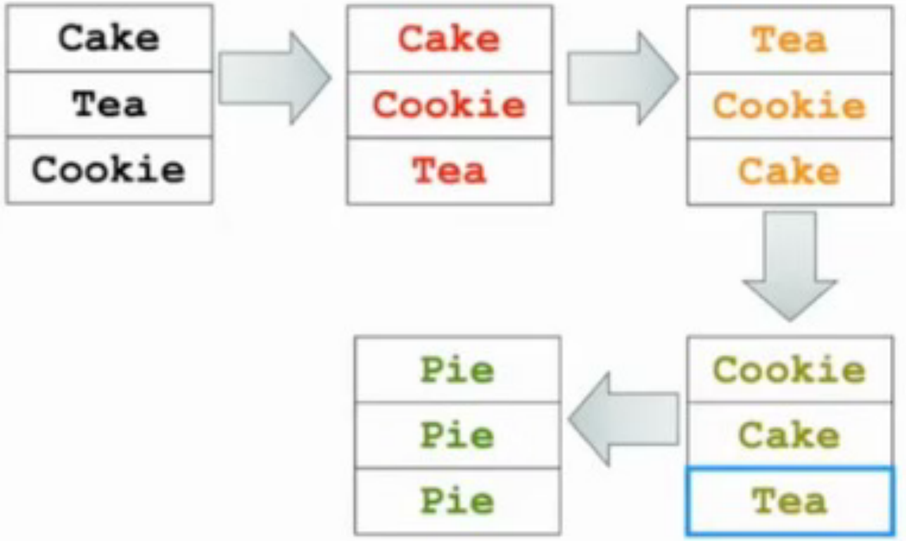
```java
List<Product> menu = new ArrayList<>();
menu.add(new Food("Cake"));
menu.add(new Drink("Tea"));
menu.add(new Food("Cookie"));

Product[] array = new Product[2];  // Array inicial con tamaño 2
array = menu.toArray(array);       // La colección ajusta el array

System.out.println(Arrays.toString(array));
```

Use `java.util.Collections` Class

Class `java.util.Collections` provides convenience methods for handling collections.

- Filling collection with values
- Searching through the collection
- Reordering collection content using:
  - `Comparable` (as implemented by objects within the array)
  - `Comparator` interface (see notes)
  - `reverse` method changes the order to opposite
  - `shuffle` method randomly reorders collection

```
Product p1 = new Food("Cake");
Product p2 = new Drink("Tea");
Product p3 = new Food("Cookie");
List<Product> menu = new ArrayList<>();
menu.add(p1);
menu.add(p2);
menu.add(p3);
Collections.sort(menu);
Collections.reverse(menu);
Collections.shuffle(menu);
Product x = Collections.binarySearch(menu, p2);
Collections.fill(menu, new Food("Pie"));
```

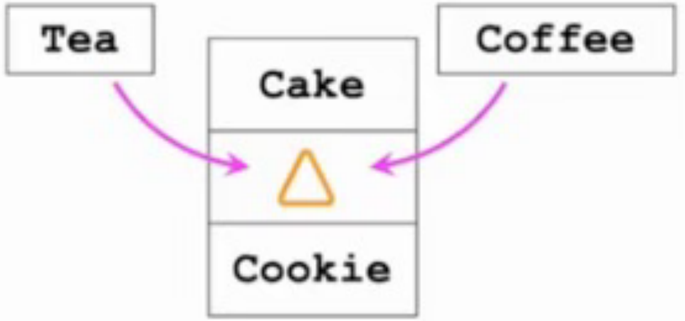


Access Collections Concurrently

Collection can be corrupted if accessed concurrently from multiple threads.

- If two or more concurrent execution paths (threads) within your program try to access the collection at the same time, they can corrupt it, if this collection is not immutable.
- Any object in a heap is not thread-safe if it is not immutable. Any thread can be interrupted, even when it is modifying an object, making other threads observe incomplete modification state.
- Making collection thread-safe does not guarantee the thread safety of the objects it contains. Only immutable objects are automatically thread-safe.

```
List<Product> list = new ArrayList<>();
/*
  Attempt to insert, update or remove elements in the
  collection concurrently, may corrupt collection.
*/
```

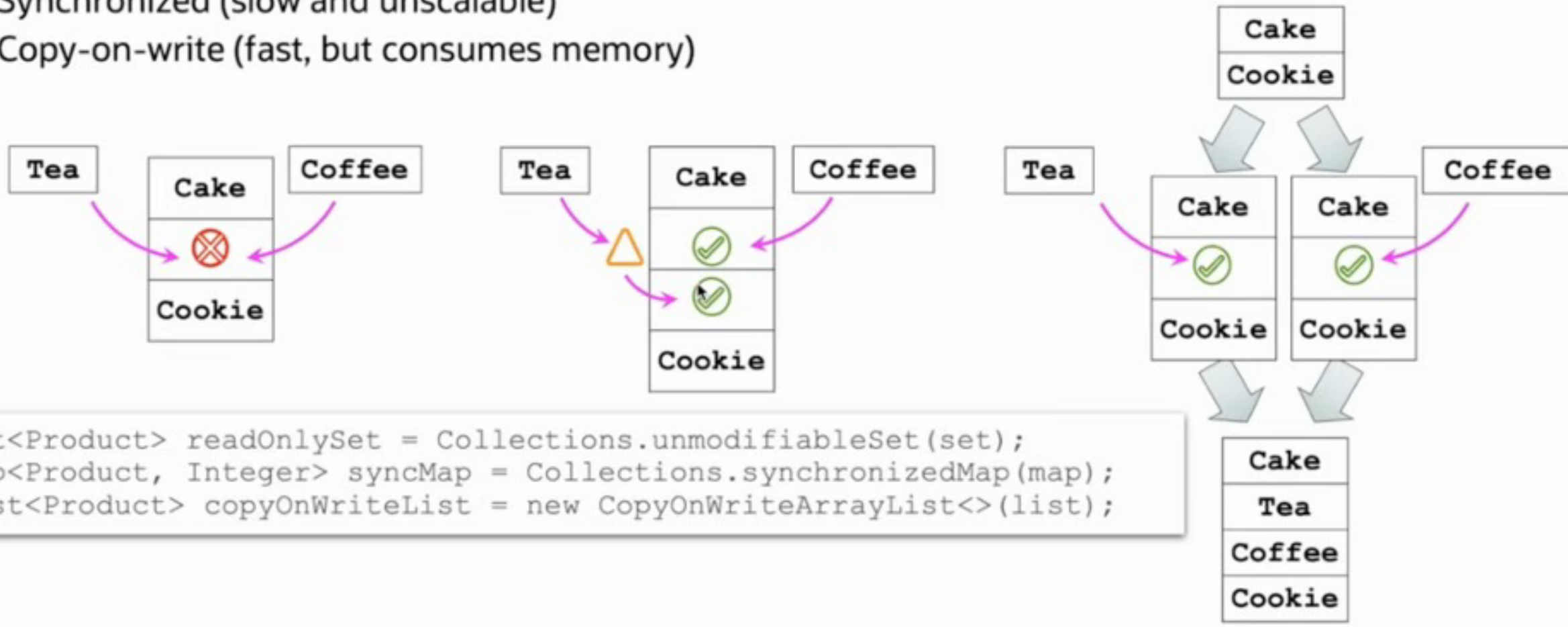


✣ Details of how to write concurrent code are covered in lesson 14 "Java Concurrency and Multithreading".

Prevent Collections Corruption

To prevent memory corruption in concurrently accessed collections, you can make collection:

- Unmodifiable (fast, but read-only)
- Synchronized (slow and unscalable)
- Copy-on-write (fast, but consumes memory)




```
Set<Product> readOnlySet = Collections.unmodifiableSet(set);
Map<Product, Integer> syncMap = Collections.synchronizedMap(map);
List<Product> copyOnWriteList = new CopyOnWriteArrayList<>(list);
```

Legacy Collection Classes

You may still encounter earlier versions of Java Collection API classes.

- Legacy collections were all defined as synchronized, which is a performance issue.
- With new collections (covered in this lesson), you can choose the type of thread-safe implementation.
- Other behaviors of old collections are almost identical to the new classes.
- Examples of equivalent collection classes:
  - Class `ArrayList` is a new equivalent of the legacy class `Vector`.
  - Class `HashMap` is a new equivalent of the legacy class `Hashtable`.

Quiz

1. Which statement about Set is true?

- ◯ Methods add and remove throw an exception when attempting to add a duplicate or remove an absent element.
- ◯ Methods add and remove will return false value when attempting to add a duplicate or remove an absent element.
- ◯ Method get returns null if the requested element is absent from the Set.
- ◯ A Set cannot be populated with a List, because a List allows duplicate elements.

Rpta: __2__

2. Which two statements about Deque are true?

- [ ] If deque is empty, pool and peek operations throw an exception.
- [ ] `offerFirst(T)` and `offerLast(T)` insert elements at the head and the tail of the deque.
- [ ] ArrayDeque objects auto-expand.
- [ ] Null values are allowed in the ArrayDeque.
- [ ] `peekFirst(T)` and `peekLast(T)` return and remove the element at the head or tail of the Deque.

Rpta: __2, __3_

3. Given:

```
Map items = new HashMap<>();
items.put(Integer.valueOf(1),"Tea");
items.put(Integer.valueOf(2),"Cake");
```

Which two statements are true?

- [ ] `items.put(Integer.valueOf(3),"Cake");` This will create an additional element with a "Cake" value.
- [ ] `items.put(Integer.valueOf(2),"Cake");` This will throw an exception.
- [ ] `items.put(Integer.valueOf(3),"Tea");` This will replace the integer key value for the "Tea" element.
- [ ] `items.put(Integer.valueOf(1),"Coffee");` This will replace "Tea" with "Coffee".

Rpta: __1, 4_

4. Which two statements are NOT true about the Java Collection API?

- [ ] All collections dynamically expand as appropriate to accommodate new elements.
- [ ] All collections provide thread-safe operations.
- [ ] All collections are all implemented internally using arrays.
- [ ] All collections allow programs to store groups of objects in memory.

Rpta: 2, 3

5. Given:

```
String[] array = {"Tea","Cake"};
List list = Arrays.asList(array);
```

Which two statements are true?

- [ ] You can replace Tea with Coffee in list.
- [ ] You can add a new String to array.
- [ ] You can replace Tea with Coffee in array.
- [ ] You can add a new String to list.

Rpta: ___1, 3_