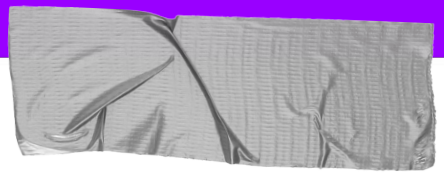
Java Certified #13

A question lead guide to prepare Java certification



Managing Concurrent Code Execution

How would you create a ConcurrentHashMap
configured to allow a maximum of 10 concurrent writer threads
and an initial capacity of 42?

Which of the following options meets this requirement?

- ➔ **var concurrentHashMap = new ConcurrentHashMap();**
- ➔ **var concurrentHashMap = new ConcurrentHashMap(42);**
- ➔ **var concurrentHashMap = new ConcurrentHashMap(42, 10);**
- ➔ **var concurrentHashMap = new ConcurrentHashMap(42, 0.88f, 10);**

var concurrentHashMap = new ConcurrentHashMap(42, 0.88f, 10);

According to the JavaDoc, the constructors are:

- **`ConcurrentHashMap`**`()`
  Creates a new, empty map with the default initial table size (16).
- **`ConcurrentHashMap`**`(int initialCapacity)`
  Creates a new, empty map with an initial table size accommodating the specified number of elements without the need to dynamically resize.
- **`ConcurrentHashMap`**`(int initialCapacity, float loadFactor)`
  Creates a new, empty map with an initial table size based on the given number of elements (`initialCapacity`) and initial table density (`loadFactor`).
- **`ConcurrentHashMap`**`(int initialCapacity, float loadFactor, int concurrencyLevel)`
  Creates a new, empty map with an initial table size based on the given number of elements (`initialCapacity`), initial table density (`loadFactor`), and number of concurrently updating threads (`concurrencyLevel`).
- **`ConcurrentHashMap`**`(`**`Map`**`<? extends `**`K`**`,? extends `**`V`**`> m)`
  Creates a new map with the same mappings as the given map.

See: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/ConcurrentHashMap.html

So the answer is:

```
var concurrentHashMap = new ConcurrentHashMap(42, 0.88f, 10);
```

Working with Arrays and Collections

Given:

```
var frenchCities = new TreeSet<String>();
frenchCities.add("Paris");
frenchCities.add("Marseille");
frenchCities.add("Lyon");
frenchCities.add("Lille");
frenchCities.add("Toulouse");
System.out.println(frenchCities.headSet( "Marseille" ));
```

What will be printed?

- ➔ [Paris]
- ➔ [Lyon, Lille, Toulouse]
- ➔ [Lille, Lyon]
- ➔ [Paris, Toulouse]
- ➔ Compilation fails

[Lille, Lyon]

headSet

public SortedSet<E> headSet(E toElement)

Description copied from interface: `NavigableSet`

Returns a view of the portion of this set whose elements are strictly less than `toElement`. The returned set is backed by this set, so changes in the returned set are reflected in this set, and vice-versa. The returned set supports all optional set operations that this set supports.

See: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeSet.html

So the answer is: [Lille, Lyon]

It is the view of the portion of this set whose elements are strictly less than Marseille, the set is sorted alphabetically: [**Lille, Lyon**, Marseille, Paris, Toulouse]

Working with Arrays and Collections

Given:

```
var ceo = new HashMap<>();
ceo.put( "Sundar Pichai", "Google" );
ceo.put( "Tim Cook", "Apple" );
ceo.put( "Mark Zuckerberg", "Meta" );
ceo.put( "Andy Jassy", "Amazon" );
```

Does the code compile?

- ➔ **True**
- ➔ **False**

True

The code compiles successfully due to Java's type inference and type erasure mechanisms.

1. The `var` keyword instructs the compiler to infer the type of the variable `ceo` at compile time. Here, the compiler determines that `ceo` is of type `HashMap<Object, Object>` because:
    - The `new HashMap<>()` uses the diamond operator (`<>`), which defers the explicit type specification.
    - The `put` method calls (`put("Sundar Pichai", "Google")`, etc.) accept `String` values for both key and value, which are subtypes of `Object`.
2. The diamond operator in `new HashMap<>()` does not perform type inference but works due to **type erasure**. Since no explicit type is provided, the compiler defaults to the raw type (`HashMap<Object, Object>`). This is not strict type inference but the result of the lack of generic type information in the context of the diamond operator.
3. The code works because Java allows such usage with `var` and the diamond operator, ensuring type compatibility at runtime.

However, note that this approach results in a `HashMap<Object, Object>`, which may not be ideal for practical use as it sacrifices type safety. Explicitly specifying the type (e.g., `HashMap<String, String>`) is usually better for clarity and maintaining type constraints.

https://bit.ly/javaOCP