You are developing an Address class. Among others, it has a field for "country". You allow only a few country codes to be used for this field. You want to keep this list of valid country codes in your class. This list can be accessed by other classes if they want to know which valid country values do you allow.

Which of the following options will you use?

You answered correctly
You had to select 1 option

```
public static final List<String> validCountryCodes = Arrays.asList( "IN", "US", "EU" );
```

This is not valid because although new elements cannot be added to this list, any one can change an existing element. For example, `validCountryCodes.set(0, "AU");`

```
public static List<String> validCountryCodes = Collections.unmodifiableList(Arrays.asList( "IN", "US", "EU" ));
```

`validCountryCodes` field should be declared final. Other than that, it is ok.

- `public static final List<String> validCountryCodes = List.of( "IN", "US", "EU" );`

```
public static final String[] validCountryCodes = new String[]{ "IN", "US", "EU" };
```

Any one can change the values of the array. For example:
`validCountryCodes[0] = "AU";`

```
public static final volatile String[] validCountryCodes = new String[]{ "IN", "US", "EU" };
```

Since the `validCountryCodes` variable itself is a constant, there is no need for volatile here anyway.

Explanation

Guideline 6-10 / MUTABLE-10: Ensure public static final field values are constants

Only immutable or unmodifiable values should be stored in public static fields. Many types are mutable and are easily overlooked, in particular arrays and collections. Mutable objects that are stored in a field whose type does not have any mutator methods can be cast back to the runtime type. Enum values should never be mutable.

In the following example, names exposes an unmodifiable view of a list in order to prevent the list from being modified.

```
import static java.util.Arrays.asList;
import static java.util.Collections.unmodifiableList;
...
public static final List<String> names = unmodifiableList(asList(
    "Fred", "Jim", "Sheila"
));
```

The of() and ofEntries() API methods, which were added in Java 9, can also be used to create unmodifiable collections:

```
public static final List<String> names =
                            List.of("Fred", "Jim", "Sheila");
```

Note that the of/ofEntries API methods return an unmodifiable collection, whereas the Collections.unmodifiable... API methods (unmodifiableCollection(), unmodifiableList(), unmodifiableMap(), etc.) return an unmodifiable view to a collection. While the collection cannot be modified via the unmodifiable view, the underlying collection may still be modified via a direct reference to it. However, the collections returned by the of/ofEntries API methods are in fact unmodifiable. See the java.util.Collections API documentation for a complete list of methods that return unmodifiable views to collections.

The copyOf methods, which were added in Java 10, can be used to create unmodifiable copies of existing collections. Unlike with unmodifiable views, if the original collection is modified the changes will not affect the unmodifiable copy. Similarly, the toUnmodifiableList(), toUnmodifiableSet(), and toUnmodifiableMap() collectors in Java 10 and later can be used to create unmodifiable collections from the elements of a stream.

Add Note