Initialization of an ArrayList in one line

Asked 15 years, 6 months ago Modified 7 months ago Viewed 3.9m times

3389

I wanted to create a list of options for testing purposes. At first, I did this:

```
ArrayList<String> places = new ArrayList<String>();
places.add("Buenos Aires");
places.add("Córdoba");
places.add("La Plata");
```

Then, I refactored the code as follows:

```
ArrayList<String> places = new ArrayList<String>(
    Arrays.asList("Buenos Aires", "Córdoba", "La Plata"));
```

Is there a better way to do this?

java collections arraylist initialization

Share
Improve this question
Follow

edited Nov 21, 2019 at 9:48
Shashanth
5,180 ● 8 ● 42 ● 52

asked Jun 17, 2009 at 4:10
Macarse
93k ● 44 ● 176 ● 232

44 If this is intended for unit testing, try groovy out for a swing. You can write your test code in it while testing java code, and use `ArrasyList<String> places = ["Buenos Aires", "Córdoba", "La Plata"]` – ripper234 Dec 31, 2010 at 21:40

5 In Java SE 7, you can substitute the parameterized type of the constructor with an empty set of type parameters (<>): Map<String, List<String>> myMap = new HashMap<>(); – Rose Jul 7, 2015 at 21:12

2 See also stackoverflow.com/questions/157944/create-arraylist-from-array – Christoffer Hammarström Oct 8, 2015 at 11:05

13 Stream.of("val1", "val2").collect(Collectors.toList()); //creates ArrayList, Java8 solution. – torina Oct 13, 2017 at 17:37

6 I don't understand why we cannot have some thing like `List<Integer> = {1, 2, 3, 4, 5}` in Java. Create and init a mutable array in one line, as I know Python and Swift could do that. – Zhou Haibo Mar 20, 2021 at 6:10

36 Answers

Sorted by: Highest score (default)

1 2 Next

2749

It would be simpler if you were to just declare it as a `List` - does it have to be an ArrayList?

```
List<String> places = Arrays.asList("Buenos Aires", "Córdoba", "La Plata");
```

Or if you have only one element:

```
List<String> places = Collections.singletonList("Buenos Aires");
```

This would mean that `places` is **immutable** (trying to change it will cause an `UnsupportedOperationException` exception to be thrown).

To make a mutable list that is a concrete `ArrayList` you can create an `ArrayList` from the immutable list:

```
ArrayList<String> places = new ArrayList<>(Arrays.asList("Buenos Aires",
"Córdoba", "La Plata"));
```

And import the correct package:

```
import java.util.Arrays;
```

Share

Improve this answer

Follow

edited Jun 12, 2022 at 11:48
Maifee Ul Asad
4,599 ● 9 ● 49 ● 115

answered Jun 17, 2009 at 4:15
Tom
44.7k ● 4 ● 43 ● 61

20 @Marcase: Can you not change your class to use a List instead of ArrayList? – L. Cornelius Dol Jun 17, 2009 at 5:04

83 As per my answer, if you're not using methods specific to `ArrayList`, it would be better design to change the declaration to `List`. Specify interfaces, not implementations. – Christoffer Hammarström Sep 9, 2010 at 12:37

10 @Christoffer Hammarström: if he changes the declaration to *List* and uses the *List<String> places = Arrays.asList(...);* he will not be able to use *places.add("blabla")* – maks Sep 25, 2011 at 12:48

73 Just to be clear, `asList(...)` returns a fixed size `List` that blows up on mutating operations like `remove` and `clear`, things the `List` contract claims to support. Even if you left declaration as `List`, you sill need to use `List l = new ArrayList(asList(...))` in order to get an object that doesn't throw

OperationNotSupported exceptions. Liskov Substitution Principle anyone? – Splash Apr 12, 2012 at 0:50

5 How can it be that I need to lookup how to initialise a fucking mutable array every time I need to? And this is supposedly the best way? What an horrible language. – Mauro Lacy Sep 14, 2023 at 7:22

2234

Actually, probably the "best" way to initialize the `ArrayList` is the method you wrote, as it does not need to create a new `List` in any way:

```
ArrayList<String> list = new ArrayList<String>();
list.add("A");
list.add("B");
list.add("C");
```

The catch is that there is quite a bit of typing required to refer to that `list` instance.

There are alternatives, such as making an anonymous inner class with an instance initializer (also known as an "double brace initialization"):

```
ArrayList<String> list = new ArrayList<String>() {{
    add("A");
    add("B");
    add("C");
}};
```

However, I'm not too fond of that method because what you end up with is a subclass of `ArrayList` which has an instance initializer, and that class is created just to create one object -- that just seems like a little bit overkill to me.

What would have been nice was if the Collection Literals proposal for Project Coin was accepted (it was slated to be introduced in Java 7, but it's not likely to be part of Java 8 either.):

```
List<String> list = ["A", "B", "C"];
```

Unfortunately it won't help you here, as it will initialize an immutable `List` rather than an `ArrayList`, and furthermore, it's not available yet, if it ever will be.

Share

Improve this answer

Follow

edited Mar 13, 2015 at 15:27

answered Jun 17, 2009 at 4:13

coobird

161k ● 35 ● 214 ● 227

189 See stackoverflow.com/questions/924285 for more information about the double-brace initialization, pros and cons. – Eddie Jun 17, 2009 at 4:21

3 you don't need the second <String>. – Delight Nov 29, 2021 at 12:31

@Delight The "diamond operator" was introduced in Java 7, which wasn't released when I wrote this answer back in 2009 ;) – coobird Jan 28, 2023 at 18:02

1146

The simple answer

Java 9 or later:

```
List<String> strings = List.of("foo", "bar", "baz");
```

`List.of(...)` will give you an *immutable* `List`, so it cannot be changed.
Which is what you want in most cases where you're prepopulating it.

This does not allow `null` elements.

Java 8 or earlier:

```
List<String> strings = Arrays.asList("foo", "bar", "baz");
```

`Arrays.asList(...)` will give you a `List` * backed by an array, so it cannot change length.
But you can call `List.set(...)`, so it's still *mutable*.

This does allow `null` elements.

* Implementation detail: It's a private nested class inside `java.util.Arrays`, named `ArrayList`, which is a different class from `java.util.ArrayList`, even though their simple names are the same.

Static import

You can make Java 8 `Arrays.asList` even shorter with a static import:

```
import static java.util.Arrays.asList;
...
List<String> strings = asList("foo", "bar", "baz");
```

Any modern IDE* will suggest and do this for you.

I don't recommend statically importing the `List.of` method as just `of`, because it's confusing.

* For example, in IntelliJ IDEA you press `Alt+Enter` and select `Static import method...`

Using `Stream`s

Why does it have to be a `List`?
With Java 8 or later you can use a [`Stream`] which is more flexible:

```
Stream<String> strings = Stream.of("foo", "bar", "baz");
```

You can concatenate `Stream`s:

```
Stream<String> strings = Stream.concat(Stream.of("foo", "bar"),
                                       Stream.of("baz", "qux"));
```

Or you can go from a `Stream` to a `List`:

```
import static java.util.stream.Collectors.toList;
...
var strings = Stream.of("foo", "bar", "baz").toList(); // Java 16

List<String> strings = Stream.of("foo", "bar", "baz").collect(toList()); //
Java 8
```

But preferably, just use the `Stream` without collecting it to a `List`.

If you specifically need a `java.util.ArrayList`*

If you want to *both* prepopulate an `ArrayList` *and* add to it afterwards, use

```
List<String> strings = new ArrayList<>(List.of("foo", "bar"));
```

or in Java 8 or earlier:

```
List<String> strings = new ArrayList<>(asList("foo", "bar"));
```

or using `Stream`:

```
import static java.util.stream.Collectors.toCollection;

List<String> strings = Stream.of("foo", "bar")
                             .collect(toCollection(ArrayList::new));
```

Then you can add to it after construction:

```
strings.add("baz");
```

But again, it's better to just use the `Stream` directly instead of collecting it to a `List`.

*You probably don't need specifically an `ArrayList`. To quote [JEP 269](JEP 269):

> There is a **small set** of use cases for initializing a mutable collection instance with a predefined set of values. It's usually preferable to have those predefined values be in an immutable collection, and then to initialize the mutable collection via a copy constructor.

(emphasis mine)

Program to interfaces, not to implementations

You said you've declared the list as an `ArrayList` in your code, but you should only do that if you're using some member of `ArrayList` that's not in `List`.

Which you are most likely not doing.

Usually you should just declare variables by the most general interface that you are going to use (e.g. `Iterable`, `Collection`, or `List`), and initialize them with the specific implementation (e.g. `ArrayList`, `LinkedList` or `Arrays.asList()`).

Otherwise you're limiting your code to that specific type, and it'll be harder to change when you want to.

For example, if you're passing an `ArrayList` to a `void method(...)`:

```
// Iterable if you just need iteration, for (String s : strings):
void method(Iterable<String> strings) {
    for (String s : strings) { ... }
}

// Collection if you also need .size(), .isEmpty(), or .stream():
void method(Collection<String> strings) {
    if (!strings.isEmpty()) { strings.stream()... }
}

// List if you also need random access, .get(index):
void method(List<String> strings) {
    strings.get(...)
}

// Don't declare a specific list implementation
// unless you're sure you need it:
void method(ArrayList<String> strings) {
    ??? // You don't want to limit yourself to just ArrayList
}
```

Another example would be always declaring variable an `InputStream` even though it is usually a `FileInputStream` or a `BufferedInputStream`, because one day soon you or somebody else will want to use some other kind of `InputStream`.

Share

Improve this answer

Follow

edited Jun 14, 2023 at 16:11

answered Sep 9, 2010 at 12:33

Christoffer Hammarström
27.8k ● 4 ● 52 ● 59

It's worth mentioning that streams need to be closed whereas lists do not. – Mike Lowery Jan 20, 2023 at 0:40

@MikeLowery: Are you thinking of `InputStream` and `OutputStream`? `java.util.stream.Stream` does not need to be closed. It does need a *terminal operation*, but you're not going to forget that, because without it the stream does nothing. – Christoffer Hammarström Jan 20, 2023 at 19:52

That's what I was actually working with, yes. But `java.util.stream.Stream` extends `BaseStream` which does have a `close()` method because it implements `AutoCloseable`. At the very least an IDE is going to complain about a potential resource leak. – Mike Lowery Jan 21, 2023 at 20:11

@MikeLowery: You don't need to close Streams in general, and i haven't seen IntelliJ IDEA complain about me not doing so. See stackoverflow.com/questions/22125169/… – Christoffer Hammarström Jan 23, 2023 at 19:00

1 Also `Arrays.asList` allows passing null elements, while `List.of` doesn't. – Renat Jun 2, 2023 at 10:06

123

If you need a simple list of size 1:

```
List<String> strings = new ArrayList<String>(Collections.singletonList("A"));
```

If you need a list of several objects:

```
List<String> strings = new ArrayList<String>();
Collections.addAll(strings,"A","B","C","D");
```

Share

Improve this answer

Follow

edited May 27, 2017 at 21:31

Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Aug 30, 2011 at 4:33

Randyaa
2,945 ● 4 ● 36 ● 50

68

With Guava you can write:

```
ArrayList<String> places = Lists.newArrayList("Buenos Aires", "Córdoba", "La
Plata");
```

In Guava there are also other useful static constructors. You can read about them here.

Share

Improve this answer

Follow

edited Dec 12, 2017 at 3:02
shmosel
50.6k ● 7 ● 77 ● 145

answered Jul 29, 2013 at 13:24
Paweł Adamski
3,405 ● 3 ● 32 ● 54

2 I'm pretty sure you can do this with just `java.util.Arrays` such as, `List<String> names = Arrays.asList("Beckah", "Sam", "Michael");` – bee Jun 26, 2015 at 19:59

3 @beckah method Arrays.asLists creates object of type List, while question is about creating ArrayList – Paweł Adamski Jul 14, 2015 at 5:34

1 *This method is not actually very useful and will likely be deprecated in the future.* – shmosel Dec 12, 2017 at 3:04

@beckah `Arrays.asList` creates a list that can't be modified. If the list needs to be modified `Lists.newArrayList` will work and `Arrays.asList` won't. – Donald Duck Aug 6, 2021 at 11:00

49

With java-9 and above, as suggested in JEP 269: Convenience Factory Methods for Collections, creating an unmodifiable `List` instead of an `ArrayList` could be achieved using *collection literals* now with -

```
List<String> list = List.of("A", "B", "C");

Set<String> set = Set.of("A", "B", "C");
```

A similar approach would apply to `Map` as well -

```
Map<String, String> map = Map.of("k1", "v1", "k2", "v2", "k3", "v3")
```

which is similar to Collection Literals proposal as stated by @coobird. Further clarified in the JEP as well -

---

Alternatives

> Language changes have been considered several times, and rejected:

> Project Coin Proposal, 29 March 2009
>
> Project Coin Proposal, 30 March 2009
>
> JEP 186 discussion on lambda-dev, January-March 2014
>
> The language proposals were set aside in preference to a library-based proposal as summarized in this message.

Related: What is the point of overloaded Convenience Factory Methods for Collections in Java 9

Share
Improve this answer
Follow

edited Mar 26 at 14:30

answered Feb 2, 2017 at 17:36
Naman
31.6k ● 30 ● 234 ● 373

Note that this creates an immutable `List` and not a modifiable `ArrayList`! – xuiqzy Mar 26 at 13:43

@xuiqzy precisely an *unmodifiable* one, if one was to mention – Naman Mar 26 at 14:31

In this case not. According to stackoverflow.com/questions/7713274/java-immutable-collections unmodifiable means the unmodifiable view of the data cannot be changed, but underlying data can be changed by other code. Here, no one has access to the underlying data and changing something through `list` or `set` itself is impossible, so it's immutable. If it was created based on a modifiable list that other code has a reference to via some variable (and not via copying the data), then it would be only unmodifiable. Fields inside the objects itself can of course always be modified. – xuiqzy Jun 5 at 20:00

Correction: For an existing list in some variable, then you get an umodifiable view list by passing it in the `Collections.unmodifiableList()` constructor , but `List.of` will always give you an "immutable" or unmodifiable list that is not a view. If you want an unmodifiable view list, you have to pass a copy of the original list to the `unmodifiableList()` constructor. According to docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/…, there are unmodifiable & unmodifiable view collections. Immutable only if all elements inside are immutable, too. – xuiqzy Jun 5 at 20:17

Oh, only `List.copyOf` is structurally unmodifiable and *not* a view collection (like `unmodifiableList` or `List.of` is), so its structure cannot be changed. Only the objects in it can change. You cannot prevent any more modification than `List.copyOf` does without making the objects itself immutable. So, in current Java terminology: `List.of` is an unmodifiable view collection and `List.copyOf` is a normal unmodifiable collection, both structurally unmodifiable via the collection object itself. I previously didn't see the constructor of `List.of` that takes a list and not single elements. – xuiqzy Jun 5 at 20:57

Collection literals didn't make it into Java 8, but it is possible to use the Stream API to initialize a list in one rather long line:

36

```
List<String> places = Stream.of("Buenos Aires", "Córdoba", "La
Plata").collect(Collectors.toList());
```

If you need to ensure that your `List` is an `ArrayList`:

```
ArrayList<String> places = Stream.of("Buenos Aires", "Córdoba", "La
Plata").collect(Collectors.toCollection(ArrayList::new));
```

Share Improve this answer Follow

answered Apr 3, 2014 at 23:21
Mark
3,267 ● 1 ● 15 ● 5

34

```
import com.google.common.collect.ImmutableList;

....

List<String> places = ImmutableList.of("Buenos Aires", "Córdoba", "La Plata");
```

Share Improve this answer Follow

answered May 12, 2010 at 13:14
George
3,453 ● 4 ● 28 ● 25

7 I don't want to add a new dependency just to do that. – Macarse May 12, 2010 at 14:16

6 That's the same as `Collections.unmodifiableList(Arrays.asList("Buenos Aires", "Córdoba", "La Plata"))`, which becomes `unmodifiableList(asList("Buenos Aires", "Córdoba", "La Plata"))` with static imports. You don't need Google Collections for this. – Christoffer Hammarström Sep 17, 2010 at 9:47

10 No, it's not the same. As ImmutableList is documenting its immutability in the result type when unmodifiableList masquerades it as a normal List. – David Pierre Nov 3, 2010 at 13:42

2 Instead of the immutable one, google collections also offer mutable array list: List<String> = Lists.newArrayList("Buenos Aires", "Córdoba", "La Plata"); – L. Holanda Aug 15, 2012 at 21:40

4 You're going to pass that `ImmutableList` to other methods that take a `List`, and then you've lost that documentation anyway. – Christoffer Hammarström Jul 16, 2014 at 13:21

26

You could create a factory method:

```
public static ArrayList<String> createArrayList(String ... elements) {
  ArrayList<String> list = new ArrayList<String>();
  for (String element : elements) {
    list.add(element);
  }
  return list;
}

....

ArrayList<String> places = createArrayList(
  "São Paulo", "Rio de Janeiro", "Brasília");
```

But it's not much better than your first refactoring.

For greater flexibility, it can be generic:

```
public static <T> ArrayList<T> createArrayList(T ... elements) {
  ArrayList<T> list = new ArrayList<T>();
  for (T element : elements) {
    list.add(element);
  }
  return list;
}
```

Share

Improve this answer

Follow

edited Mar 4, 2018 at 21:29

Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered May 4, 2010 at 0:57

Jordão
56.4k ● 13 ● 116 ● 146

2 Look back at the original post, it is asking for array initialization in *one line*, not 7 additional lines. – L. Holanda Aug 15, 2012 at 21:24

8 @LeoHolanda: Creating factory methods for every little thing is too much, I agree. But *depending* on the situation, and on the number of times that that method is going to be used, it might make sense to create it. Creating extra abstraction layers is meant to *remove* complexity, by creating more *meaningful* methods that capture the *intent* of the designer. – Jordão Aug 15, 2012 at 22:39

I think we can replace the enhanced `for` with `Collections.addAll(elements)` as described here. – ggorlen Jul 28, 2021 at 23:01

14

In Java 9 we can easily initialize an `ArrayList` in a single line:

```
List<String> places = List.of("Buenos Aires", "Córdoba", "La Plata");
```

or

```
List<String> places = new ArrayList<>(List.of("Buenos Aires", "Córdoba", "La
Plata"));
```

This new approach of Java 9 has many advantages over the previous ones:

1. Space Efficiency
2. Immutability
3. Thread Safe

See this post for more details -> *What is the difference between List.of and Arrays.asList?*

Share
Improve this answer
Follow

edited Mar 4, 2018 at 22:29
Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Oct 6, 2017 at 1:16
Mohit Tyagi
2,876 ● 4 ● 19 ● 30

The first example does not create an `ArrayList`, despite the preceding sentence indicating that it does. – M. Justin Sep 10, 2023 at 20:28

14

Java 9 has the following method to create an *immutable* list as documented (or documented as *unmodifiable* Java 10+):

```
List<String> places = List.of("Buenos Aires", "Córdoba", "La Plata");
```

which is easily adapted to create a mutable list, if required:

```
List<String> places = new ArrayList<>(List.of("Buenos Aires", "Córdoba", "La
Plata"));
```

Similar methods are available for `Set` and `Map`.

Please note that these methods do not accept `null` elements, a little bit hidden in the documentation:

> Throws: `NullPointerException` - if an element is `null` ...

Share
Improve this answer
Follow

edited Dec 3, 2023 at 7:48

answered Dec 6, 2016 at 9:57
user85421
29.6k ● 11 ● 65 ● 92

1 Its good that you explicitly said "immutable list" and then showed another example of mutable list because it makes it clear which to use when. – Tito Jun 7, 2019 at 14:34

Simply use below code as follows.

10

```
List<String> list = new ArrayList<String>() {{
            add("A");
            add("B");
            add("C");
}};
```

Share
Improve this answer
Follow

edited Dec 12, 2017 at 2:57
KeLiuyue
8,217 ● 4 ● 27 ● 43

answered Sep 21, 2013 at 9:43
user2801794
127 ● 1 ● 3

This is not one-line – ZhekaKozlov Feb 19 at 11:17

8

About the most compact way to do this is:

```
Double array[] = { 1.0, 2.0, 3.0};
List<Double> list = Arrays.asList(array);
```

Share Improve this answer Follow

answered Dec 15, 2014 at 11:44
Richard B
935 ● 1 ● 14 ● 42

8

Here is another way:

```
List<String> values = Stream.of("One", "Two").collect(Collectors.toList());
```

Share
Improve this answer
Follow

edited Mar 1, 2017 at 0:49

answered Feb 28, 2017 at 22:50
Henok T
1,084 ● 13 ● 8

8

With Eclipse Collections you can write the following:

```
List<String> list = Lists.mutable.with("Buenos Aires", "Córdoba", "La Plata");
```

You can also be more specific about the types and whether they are Mutable or Immutable.

```
MutableList<String> mList = Lists.mutable.with("Buenos Aires", "Córdoba", "La
Plata");
ImmutableList<String> iList = Lists.immutable.with("Buenos Aires", "Córdoba",
"La Plata");
```

You can also do the same with Sets and Bags:

```
Set<String> set = Sets.mutable.with("Buenos Aires", "Córdoba", "La Plata");
MutableSet<String> mSet = Sets.mutable.with("Buenos Aires", "Córdoba", "La
Plata");
ImmutableSet<String> iSet = Sets.immutable.with("Buenos Aires", "Córdoba", "La
Plata");

Bag<String> bag = Bags.mutable.with("Buenos Aires", "Córdoba", "La Plata");
MutableBag<String> mBag = Bags.mutable.with("Buenos Aires", "Córdoba", "La
Plata");
ImmutableBag<String> iBag = Bags.immutable.with("Buenos Aires", "Córdoba", "La
Plata");
```

Note: I am a committer for Eclipse Collections.

Share

Improve this answer

Follow

edited May 1, 2019 at 1:53

answered Jan 28, 2015 at 1:30

Donald Raab

6,676 ● 2 ● 38 ● 45

8

(Should be a comment, but too long, so new reply). As others have mentioned, the `Arrays.asList` method is fixed size, but that's not the only issue with it. It also doesn't handle inheritance very well. For instance, suppose you have the following:

```
class A {}
class B extends A {}

public List<A> getAList() {
    return Arrays.asList(new B());
}
```

The above results in a compiler error, because `List<B>` (which is what is returned by `Arrays.asList`) is not a subclass of `List<A>`, even though you can add Objects of type `B` to a `List<A>` object. To get around this, you need to do something like:

```
new ArrayList<A>(Arrays.<A>asList(b1, b2, b3))
```

This is probably the best way to go about doing this, especially if you need an unbounded list or need to use inheritance.

Share

Improve this answer

Follow

edited Nov 18, 2023 at 3:37

Benjamin Loison

5,582 ● 4 ● 19 ● 37

answered Jun 6, 2013 at 5:44

user439407

1,746 ● 2 ● 21 ● 42

this is relevant only for Java versions previous to Java 8 (or even older) - the mentioned compiler error cannot be reproduced with Java 8 or later (example ideone.com/WHPY5d) -

with not-so-old versions, the compiler is able to correctly infer the list type – user85421 Nov 23, 2023 at 8:35

6

You can use the below statements:

Code Snippet:

```
String [] arr = {"Sharlock", "Homes", "Watson"};

List<String> names = Arrays.asList(arr);
```

Share
Improve this answer
Follow

edited Jun 20, 2020 at 9:12

community wiki
2 revs, 2 users 80%
rashedcs

2 You can inline first expression to have compact solution: `letters = Arrays.asList(new String[]{"A", "B", "C"});` – Pavel Repin Feb 26, 2018 at 13:42

6

```
List<String> names = Arrays.asList("2", "@2234", "21", "11");
```

Share
Improve this answer
Follow

edited Jul 5, 2023 at 12:12
Dmitriy Popov
2,350 ● 3 ● 26 ● 38

answered Jun 22, 2014 at 8:12
Ran Adler
3,711 ● 31 ● 28

Note that this creates an immutable List and not a modifiable ArrayList! – xuiqzy Mar 26 at 13:44

6

There are multiple ways to create and initialize list in one line. Some examples:

```
//Using Double brace initialization, creates a new (anonymous) subclass of
ArrayList
List<String> list1 = new ArrayList<>() {{ add("A");  add("B"); }};

//Immutable List
List<String> list2 = List.of("A", "B");

//Fixed size list. Can't add or remove element, though replacing the element is
allowed.
List<String> list3 = Arrays.asList("A", "B");
```

```
//Modifiable list
List<String> list4 = new ArrayList<>(Arrays.asList("A", "B"));

//Using Java Stream, no guarantees on the type, mutability, serializability, or
thread-safety
List<String> list5 = Stream.of("A", "B").collect(Collectors.toList());

//Thread safe List
List<String> list6 = new CopyOnWriteArrayList<>(Arrays.asList("A", "B"));
```

Share

Improve this answer

Follow

edited Nov 23, 2023 at 8:01

user85421

29.6k ● 11 ● 65 ● 92

answered Dec 27, 2022 at 14:05

Ajay Singh

491 ● 1 ● 8 ● 21

5

Like Tom said:

```
List<String> places = Arrays.asList("Buenos Aires", "Córdoba", "La Plata");
```

But since you complained of wanting an ArrayList, you should first know that ArrayList is a subclass of List and you could simply add this line:

```
ArrayList<String> myPlaces = new ArrayList(places);
```

Although, that might make you complain about 'performance'.

In that case, it doesn't make sense to me, why, since your list is predefined it wasn't defined as an array (since the size is known at the time of initialization). And if that's an option for you:

```
String[] places = { "Buenos Aires", "Córdoba", "La Plata" };
```

In case you don't care about the minor performance differences then you can also copy an array to an ArrayList very simply:

```
ArrayList<String> myPlaces = new ArrayList(Arrays.asList(places));
```

Okay, but in the future, you need a bit more than just the place name, you need a country code too. Assuming this is still a predefined list that will never change during run-time, then it's fitting to use an `enum` set, which would require re-compilation if the list needed to be changed in the future.

```
enum Places { BUENOS_AIRES, CORDOBA, LA_PLATA }
```

would become:

```
enum Places {
    BUENOS_AIRES("Buenos Aires", 123),
    CORDOBA("Córdoba", 456),
    LA_PLATA("La Plata", 789);

    String name;
    int code;

    Places(String name, int code) {
      this.name = name;
      this.code = code;
    }
}
```

Enums have a static `values` method that returns an array containing all of the values of the enum in the order they are declared, e.g.:

```
for (Places p : Places.values()) {
    System.out.printf("The place %s has code %d%n",
                  p.name, p.code);
}
```

In that case, I guess you wouldn't need your ArrayList.

P.S. Randyaa demonstrated another nice way using the static utility method Collections.addAll.

Share
Improve this answer
Follow

edited Jul 5, 2023 at 12:11
Dmitriy Popov
2,350 ●3 ●26 ●38

answered Jun 17, 2013 at 17:57
Ozzy
8,312 ●7 ●57 ●96

4

You can use `StickyList` from Cactoos:

```
List<String> names = new StickyList<>(
  "Scott Fitzgerald", "Fyodor Dostoyevsky"
);
```

Share Improve this answer Follow

answered Jun 25, 2017 at 17:48
yegor256
105k ●130 ●460 ●620

4

Using `Arrays.asList("Buenos Aires", "Córdoba", "La Plata");` is correct. but Any calls to `Arrays.asList()` with zero arguments or only one argument could be replaced with either a call to `Collections.singletonList()` or `Collections.emptyList()` which will save some memory.

Note: the list returned by `Collections.singletonList()` is immutable, while the list returned `Arrays.asList()` allows calling the `set()` method. This may break the code in rare cases.

Share
Improve this answer
Follow

edited Jul 5, 2023 at 12:12
Dmitriy Popov
2,350 ● 3 ● 26 ● 38

answered May 19, 2021 at 11:35
Akila Sachitra
1,266 ● 2 ● 16 ● 25

4

Yes. With the help of `Arrays` you can initialize an immutable `List` (instead of an `ArrayList`) in one line:

```
List<String> strlist = Arrays.asList("aaa", "bbb", "ccc");
```

Share
Improve this answer
Follow

edited Mar 27 at 14:26
FWDekker
2,819 ● 2 ● 22 ● 30

answered Sep 13, 2016 at 5:54
Akash Manngroliya
181 ● 1 ● 6

4

You can use of() method from java 9 onwards

```
List<String> places = List.of("Buenos Aires","Córdoba","La Plata");
```

of() methods create immutable list. If you want to modifie you got exception.

Share
Improve this answer
Follow

edited May 13 at 17:34
dbc
116k ● 25 ● 261 ● 382

answered May 13 at 16:09
Amar kumar Nayak
84 ● 10

3

Try with this code line:

```
Collections.singletonList(provider)
```

Share
Improve this answer
Follow

edited May 16, 2016 at 14:25
Gustavo Morales
2,674 ● 9 ● 32 ● 39

answered May 16, 2016 at 13:45
Ant20
2,253 ● 2 ● 13 ● 16

1 Add a brief description of your answer. – Gustavo Morales May 16, 2016 at 14:05

2

In Java, you can't do

```
ArrayList<String> places = new ArrayList<String>( Arrays.asList("Buenos Aires",
"Córdoba", "La Plata"));
```

As was pointed out, you'd need to do a double brace initialization:

```
List<String> places = new ArrayList<String>() {{ add("x"); add("y"); }};
```

But this may force you into adding an annotation `@SuppressWarnings("serial")` or generate a serial UUID which is annoying. Also most code formatters will unwrap that into multiple statements/lines.

Alternatively you can do

```
List<String> places = Arrays.asList(new String[] {"x", "y" });
```

but then you may want to do a `@SuppressWarnings("unchecked")`.

Also according to javadoc you should be able to do this:

```
List<String> stooges = Arrays.asList("Larry", "Moe", "Curly");
```

But I'm not able to get it to compile with JDK 1.6.

Share
Improve this answer
Follow

edited Feb 13, 2012 at 22:44
DaveShaw
52.7k ● 17 ● 117 ● 142

answered Feb 13, 2012 at 21:47
Dawg
61 ● 1

5 Wrong! You *can* do the first line, and that is the right answer btw – Bohemian ♦ Aug 9, 2012 at 12:53

1

```
Collections.singletonList(messageBody)
```

If you'd need to have a list of **one item**!

Collections is from *java.util* package.

Share Improve this answer Follow

answered Apr 3, 2015 at 9:30
ViliusK
11.5k ● 4 ● 70 ● 72

1

interestingly no one-liner with the other overloaded `Stream::collect` *method is listed*

```
ArrayList<String> places = Stream.of( "Buenos Aires", "Córdoba", "La Plata"
).collect( ArrayList::new, ArrayList::add, ArrayList::addAll );
```

Share Improve this answer Follow

answered Oct 4, 2019 at 15:04
Kaplan
3,663 ● 18 ● 22

Totally different answer. Thanks – user20072008 Mar 6, 2022 at 22:43

1

Simplest way : you can use this approach to add multiple elements to any type of collection like ArrayList and HashSet

```
ArrayList<String> allViews = new ArrayList<String>();
Collections.addAll(allViews,"hello","world","abc","def","ghi");
```

Share Improve this answer Follow

answered Feb 13, 2022 at 5:55
PRANAV SINGH
1,190 ● 13 ● 22

1

The best way to do it:

```
package main_package;

import java.util.ArrayList;


public class Stackkkk {
    public static void main(String[] args) {
        ArrayList<Object> list = new ArrayList<Object>();
        add(list, "1", "2", "3", "4", "5", "6");
        System.out.println("I added " + list.size() + " element in one line");
    }

    public static void add(ArrayList<Object> list, Object... objects) {
        for (Object object : objects)
            list.add(object);
    }
}
```

Just create a function that can have as many elements as you want and call it to add them in one line.

Share

Improve this answer

Follow

edited Jul 5, 2023 at 12:14
Dmitriy Popov
2,350 ● 3 ● 26 ● 38

answered Jul 17, 2016 at 18:31
Charif DZ
14.7k ● 3 ● 24 ● 42

2 If you go through all the trouble you might as well make it a template method instead of using plain `Object`. – Robert Sep 8, 2016 at 21:27

1 2 Next

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.