@NoArgsConstructor, @RequiredArgsConstructor, @AllArgsConstructor

Constructors made to order: Generates constructors that take no arguments, one argument per final / non-null field, or one argument for every field.

Overview

This set of 3 annotations generate a constructor that will accept 1 parameter for certain fields, and simply assigns this parameter to the field.

`@NoArgsConstructor` will generate a constructor with no parameters. If this is not possible (because of final fields), a compiler error will result instead, unless `@NoArgsConstructor(force = true)` is used, then all final fields are initialized with `0` / `false` / `null`. For fields with constraints, such as `@NonNull` fields, *no* check is generated,so be aware that these constraints will generally not be fulfilled until those fields are properly initialized later. Certain java constructs, such as hibernate and the Service Provider Interface require a no-args constructor. This annotation is useful primarily in combination with either `@Data` or one of the other constructor generating annotations.

`@RequiredArgsConstructor` generates a constructor with 1 parameter for each field that requires special handling. All non-initialized `final` fields get a parameter, as well as any fields that are marked as `@NonNull` that aren't initialized where they are declared. For those fields marked with `@NonNull`, an explicit null check is also generated. The constructor will throw a `NullPointerException` if any of the parameters intended for the fields marked with `@NonNull` contain `null`. The order of the parameters match the order in which the fields appear in your class.

`@AllArgsConstructor` generates a constructor with 1 parameter for each field in your class. Fields marked with `@NonNull` result in null checks on those parameters.

Each of these annotations allows an alternate form, where the generated constructor is always private, and an additional static factory method that wraps around the private constructor is generated. This mode is enabled by supplying the `staticName` value for the annotation, like so: `@RequiredArgsConstructor(staticName="of")`. Such a static factory method will infer generics, unlike a normal constructor. This means your API users get write `MapEntry.of("foo", 5)` instead of the much longer `new MapEntry<String, Integer>("foo", 5)`.

To put annotations on the generated constructor, you can use `onConstructor=@__({@AnnotationsHere})`, but be careful; this is an experimental feature. For more details see the documentation on the onX feature.

Static fields are skipped by these annotations.

Unlike most other lombok annotations, the existence of an explicit constructor does not stop these annotations from generating their own constructor. This means you can write your own specialized constructor, and let lombok generate the boilerplate ones as well. If a conflict arises (one of your constructors ends up with the same signature as one that lombok generates), a compiler error will occur.

With Lombok

```
import lombok.AccessLevel;
import lombok.RequiredArgsConstructor;
import lombok.AllArgsConstructor;
import lombok.NonNull;

@RequiredArgsConstructor(staticName = "of")
@AllArgsConstructor(access = AccessLevel.PROTECTED)
public class ConstructorExample<T> {
  private int x, y;
  @NonNull private T description;

  @NoArgsConstructor
  public static class NoArgsExample {
    @NonNull private String field;
  }
}
```

Vanilla Java

```
public class ConstructorExample<T> {
  private int x, y;
  @NonNull private T description;

  private ConstructorExample(T description) {
    if (description == null) throw new NullPointerException("description");
    this.description = description;
  }

  public static <T> ConstructorExample<T> of(T description) {
    return new ConstructorExample<T>(description);
  }

  @java.beans.ConstructorProperties({"x", "y", "description"})
  protected ConstructorExample(int x, int y, T description) {
    if (description == null) throw new NullPointerException("description");
    this.x = x;
    this.y = y;
    this.description = description;
  }

  public static class NoArgsExample {
    @NonNull private String field;

    public NoArgsExample() {
    }
  }
}
```

Supported configuration keys:

`lombok.anyConstructor.addConstructorProperties` = [`true` | `false`] **(default: `false`)**
If set to `true`, then lombok will add a `@java.beans.ConstructorProperties` to generated constructors.
`lombok.` [`allArgsConstructor` | `requiredArgsConstructor` | `noArgsConstructor`] `.flagUsage` = [`warning` | `error`] **(default: not set)**
Lombok will flag any usage of the relevant annotation (`@AllArgsConstructor`, `@RequiredArgsConstructor` or `@NoArgsConstructor`) as a warning or error if configured.
`lombok.anyConstructor.flagUsage` = [`warning` | `error`] **(default: not set)**
Lombok will flag any usage of any of the 3 constructor-generating annotations as a warning or error if configured.
`lombok.copyableAnnotations` = [***A list of fully qualified types***] **(default: empty list)**
Lombok will copy any of these annotations from the field to the constructor parameter, the setter parameter, and the getter method. Note that lombok ships with a bunch of annotations 'out of the box' which are known to be copyable: All popular nullable/nonnull annotations.
`lombok.noArgsConstructor.extraPrivate` = [`true` | `false`] **(default: false)**
If `true`, lombok will generate a private no-args constructor for any `@Value` or `@Data` annotated class, which sets all fields to default values (null / 0 / false).

Small print

Even if a field is explicitly initialized with `null`, lombok will consider the requirement to avoid null as fulfilled, and will *NOT* consider the field as a 'required' argument. The assumption is that if you explicitly assign `null` to a field that you've also marked as `@NonNull` signals you must know what you're doing.

The `@java.beans.ConstructorProperties` annotation is never generated for a constructor with no arguments. This also explains why `@NoArgsConstructor` lacks the `suppressConstructorProperties` annotation method. The generated static factory methods also do not get `@ConstructorProperties`, as this annotation can only be added to real constructors.

`@XArgsConstructor` can also be used on an enum definition. The generated constructor will always be private, because non-private constructors aren't legal in enums. You don't have to specify `AccessLevel.PRIVATE`.

Various well known annotations about nullity cause null checks to be inserted and will be copied to the parameter. See Getter/Setter documentation's small print for more information.

The `flagUsage` configuration keys do not trigger when a constructor is generated by `@Data`, `@Value` or any other lombok annotation.