@Builder

... and Bob's your uncle: No-hassle fancy-pants APIs for object creation!

`@Builder` was introduced as experimental feature in lombok v0.12.0.

`@Builder` gained `@Singular` support and was promoted to the main `lombok` package since lombok v1.16.0.

`@Builder` with `@Singular` adds a clear method since lombok v1.16.8.

`@Builder.Default` functionality was added in lombok v1.16.16.

`@Builder(builderMethodName = "")` is legal (and will suppress generation of the builder method) starting with lombok v1.18.8.

`@Builder(access = AccessLevel.PACKAGE)` is legal (and will generate the builder class, the builder method, etc with the indicated access level) starting with lombok v1.18.8.

Overview

The `@Builder` annotation produces complex builder APIs for your classes.

`@Builder` lets you automatically produce the code required to have your class be instantiable with code such as:

```
Person.builder()
    .name("Adam Savage")
    .city("San Francisco")
    .job("Mythbusters")
    .job("Unchained Reaction")
    .build();
```

`@Builder` can be placed on a class, or on a constructor, or on a method. While the "on a class" and "on a constructor" mode are the most common use-case, `@Builder` is most easily explained with the "method" use-case.

A method annotated with `@Builder` (from now on called the *target*) causes the following 7 things to be generated:

- An inner static class named `FooBuilder`, with the same type arguments as the static method (called the *builder*).
- In the *builder*: One private non-static non-final field for each parameter of the *target*.
- In the *builder*: A package private no-args empty constructor.
- In the *builder*: A 'setter'-like method for each parameter of the *target*: It has the same type as that parameter and the same name. It returns the builder itself, so that the setter calls can be chained, as in the above example.
- In the *builder*: A `build()` method which calls the method, passing in each field. It returns the same type that the *target* returns.
- In the *builder*: A sensible `toString()` implementation.
- In the class containing the *target*: A `builder()` method, which creates a new instance of the *builder*.

Each listed generated element will be silently skipped if that element already exists (disregarding parameter counts and looking only at names). This includes the *builder* itself: If that class already exists, lombok will simply start injecting fields and methods inside this already existing class, unless of course the fields / methods to be injected already exist. You may not put any other method (or constructor) generating lombok annotation on a builder class though; for example, you can not put `@EqualsAndHashCode` on the builder class.

`@Builder` can generate so-called 'singular' methods for collection parameters/fields. These take 1 element instead of an entire list, and add the element to the list. For example:

```
Person.builder()
    .job("Mythbusters")
    .job("Unchained Reaction")
    .build();
```

would result in the `List<String> jobs` field to have 2 strings in it. To get this behavior, the field/parameter needs to be annotated with `@Singular`. The feature has its own documentation.

Now that the "method" mode is clear, putting a `@Builder` annotation on a constructor functions similarly; effectively, constructors are just static methods that have a special syntax to invoke them: Their 'return type' is the class they construct, and their type parameters are the same as the type parameters of the class itself.

Finally, applying `@Builder` to a class is as if you added `@AllArgsConstructor(access = AccessLevel.PACKAGE)` to the class and applied the `@Builder` annotation to this all-args-constructor. This only works if you haven't written any explicit constructors yourself or allowed lombok to create one such as with `@NoArgsConstructor`. If you do have an explicit constructor, put the `@Builder` annotation on the constructor instead of on the class. Note that if you put both `@Value` and `@Builder` on a class, the package-private constructor that `@Builder` wants to generate 'wins' and suppresses the constructor that `@Value` wants to make.

If using `@Builder` to generate builders to produce instances of your own class (this is always the case unless adding `@Builder` to a method that doesn't return your own type), you can use `@Builder(toBuilder = true)` to also generate an instance method in your class called `toBuilder()`; it creates a new builder that starts out with all the values of this instance. You can put the `@Builder.ObtainVia` annotation on the parameters (in case of a constructor or method) or fields (in case of `@Builder` on a type) to indicate alternative means by which the value for that field/parameter is obtained from this instance. For example, you can specify a method to be invoked: `@Builder.ObtainVia(method = "calculateFoo")`.

The name of the builder class is `FoobarBuilder`, where *Foobar* is the simplified, title-cased form of the return type of the *target* - that is, the name of your type for `@Builder` on constructors and types, and the name of the return type for `@Builder` on methods. For example, if `@Builder` is applied to a class named `com.yoyodyne.FancyList<T>`, then the builder name will be `FancyListBuilder<T>`. If `@Builder` is applied to a method that returns `void`, the builder will be named `VoidBuilder`.

The configurable aspects of builder are:

- The *builder's class name* (default: return type + 'Builder')
- The *build()* method's name (default: `"build"`)
- The *builder()* method's name (default: `"builder"`)
- If you want `toBuilder()` (default: no)
- The access level of all generated elements (default: `public`).
- (discouraged) If you want your builder's 'set' methods to have a prefix, i.e. `Person.builder().setName("Jane").build()` instead of `Person.builder().name("Jane").build()` and what it should be.

Example usage where all options are changed from their defaults:
`@Builder(builderClassName = "HelloWorldBuilder", buildMethodName = "execute", builderMethodName = "helloWorld", toBuilder = true, access = AccessLevel.PRIVATE, setterPrefix = "set")`

Looking to use your builder with Jackson, the JSON/XML tool? We have you covered: Check out the @Jacksonized feature.

@Builder.Default

If a certain field/parameter is never set during a build session, then it always gets 0 / `null` / false. If you've put `@Builder` on a class (and not a method or constructor) you can instead specify the default directly on the field, and annotate the field with `@Builder.Default`:
`@Builder.Default private final long created = System.currentTimeMillis();`
Calling Lombok-generated constructors such as `@NoArgsConstructor` will also make use of the defaults specified using `@Builder.Default` however explicit constructors will no longer use the default values and will need to be set manually or call a Lombok-generated constructor such as `this();` to set the defaults.

@Singular

By annotating one of the parameters (if annotating a method or constructor with `@Builder`) or fields (if annotating a class with `@Builder`) with the `@Singular` annotation, lombok will treat that builder node as a collection, and it generates 2 'adder' methods instead of a 'setter' method. One which adds a single element to the collection, and one which adds all elements of another collection to the collection. No setter to just set the collection (replacing whatever was already added) will be generated. A 'clear' method is also generated. These 'singular' builders are very complicated in order to guarantee the following properties:

- When invoking `build()`, the produced collection will be immutable.
- Calling one of the 'adder' methods, or the 'clear' method, after invoking `build()` does not modify any already generated objects, and, if `build()` is later called again, another collection with all the elements added since the creation of the builder is generated.
- The produced collection will be compacted to the smallest feasible format while remaining efficient.

`@Singular` can only be applied to collection types known to lombok. Currently, the supported types are:

- `java.util`:
  - `Iterable`, `Collection`, and `List` (backed by a compacted unmodifiable `ArrayList` in the general case).
  - `Set`, `SortedSet`, and `NavigableSet` (backed by a smartly sized unmodifiable `HashSet` or `TreeSet` in the general case).
  - `Map`, `SortedMap`, and `NavigableMap` (backed by a smartly sized unmodifiable `HashMap` or `TreeMap` in the general case).
- Guava's `com.google.common.collect`:
  - `ImmutableCollection` and `ImmutableList` (backed by the builder feature of `ImmutableList`).
  - `ImmutableSet` and `ImmutableSortedSet` (backed by the builder feature of those types).
  - `ImmutableMap`, `ImmutableBiMap`, and `ImmutableSortedMap` (backed by the builder feature of those types).
  - `ImmutableTable` (backed by the builder feature of `ImmutableTable`).

If your identifiers are written in common english, lombok assumes that the name of any collection with `@Singular` on it is an english plural and will attempt to automatically singularize that name. If this is possible, the add-one method will use this name. For example, if your collection is called `statuses`, then the add-one method will automatically be called `status`. You can also specify the singular form of your identifier explicitly by passing the singular form as argument to the annotation like so: `@Singular("axis") List<Line> axes;`.
If lombok cannot singularize your identifier, or it is ambiguous, lombok will generate an error and force you to explicitly specify the singular name.

The snippet below does not show what lombok generates for a `@Singular` field/parameter because it is rather complicated. You can view a snippet here.

If also using `setterPrefix = "with"`, the generated names are, for example, `withName` (add 1 name), `withNames` (add many names), and `clearNames` (reset all names).

Ordinarily, the generated 'plural form' method (which takes in a collection, and adds each element in this collection) will check if a `null` is passed the same way `@NonNull` does (by default, throws a `NullPointerException` with an appropriate message). However, you can also tell lombok to ignore such collection (so, add nothing, return immediately): `@Singular(ignoreNullCollections = true`.

With Jackson

You can customize parts of your builder, for example adding another method to the builder class, or annotating a method in the builder class, by making the builder class yourself. Lombok will generate everything that you do not manually add, and put it into this builder class. For example, if you are trying to configure jackson to use a specific subtype for a collection, you can write something like:

```
@Value @Builder
@JsonDeserialize(builder = JacksonExample.JacksonExampleBuilder.class)
public class JacksonExample {
	@Singular(nullBehavior = NullCollectionBehavior.IGNORE) private List<Foo> foos;

	@JsonPOJOBuilder(withPrefix = "")
	public static class JacksonExampleBuilder implements JacksonExampleBuilderMeta {
	}

	private interface JacksonExampleBuilderMeta {
		@JsonDeserialize(contentAs = FooImpl.class) JacksonExampleBuilder foos(List<? exte
	}
}
```

With Lombok

```
import lombok.Builder;
import lombok.Singular;
import java.util.Set;

@Builder
public class BuilderExample {
  @Builder.Default private long created = System.currentTimeMillis();
  private String name;
  private int age;
  @Singular private Set<String> occupations;
}
```

Vanilla Java

```
import java.util.Set;

public class BuilderExample {
  private long created;
  private String name;
  private int age;
  private Set<String> occupations;

  BuilderExample(String name, int age, Set<String> occupations) {
    this.name = name;
    this.age = age;
    this.occupations = occupations;
  }

  private static long $default$created() {
    return System.currentTimeMillis();
  }

  public static BuilderExampleBuilder builder() {
    return new BuilderExampleBuilder();
  }

  public static class BuilderExampleBuilder {
    private long created;
    private boolean created$set;
    private String name;
    private int age;
    private java.util.ArrayList<String> occupations;

    BuilderExampleBuilder() {
    }

    public BuilderExampleBuilder created(long created) {
      this.created = created;
      this.created$set = true;
      return this;
    }

    public BuilderExampleBuilder name(String name) {
      this.name = name;
      return this;
    }

    public BuilderExampleBuilder age(int age) {
      this.age = age;
      return this;
    }

    public BuilderExampleBuilder occupation(String occupation) {
      if (this.occupations == null) {
        this.occupations = new java.util.ArrayList<String>();
      }

      this.occupations.add(occupation);
      return this;
    }

    public BuilderExampleBuilder occupations(Collection<? extends String> occupations) {
      if (this.occupations == null) {
        this.occupations = new java.util.ArrayList<String>();
      }

      this.occupations.addAll(occupations);
      return this;
    }

    public BuilderExampleBuilder clearOccupations() {
      if (this.occupations != null) {
        this.occupations.clear();
      }

      return this;
    }

    public BuilderExample build() {
      // complicated switch statement to produce a compact properly sized immutable set omitted.
      Set<String> occupations = ...;
      return new BuilderExample(created$set ? created : BuilderExample.$default$created(), name, age, occup
    }

    @java.lang.Override
    public String toString() {
      return "BuilderExample.BuilderExampleBuilder(created = " + this.created + ", name = " + this.name + "
    }
  }
}
```

Supported configuration keys:

`lombok.builder.className` = [a java identifier with an optional star to indicate where the return type name goes] (default: `*Builder`)
Unless you explicitly pick the builder's class name with the `builderClassName` parameter, this name is chosen; any star in the name is replaced with the relevant return type.
`lombok.builder.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@Builder` as a warning or error if configured.
`lombok.singular.useGuava` = [`true` | `false`] (default: false)
If `true`, lombok will use guava's `ImmutableXxx` builders and types to implement `java.util` collection interfaces, instead of creating implementations based on `Collections.unmodifiableXxx`. You must ensure that guava is actually available on the classpath and buildpath if you use this setting. Guava is used automatically if your field/parameter has one of the guava `ImmutableXxx` types.
`lombok.singular.auto` = [`true` | `false`] (default: true)
If `true` (which is the default), lombok automatically tries to singularize your identifier name by assuming that it is a common english plural. If `false`, you must always explicitly specify the singular name, and lombok will generate an error if you don't (useful if you write your code in a language other than english).

Small print

@Singular support for `java.util.NavigableMap/Set` only works if you are compiling with JDK1.8 or higher.

You cannot manually provide some or all parts of a `@Singular` node; the code lombok generates is too complex for this. If you want to manually control (part of) the builder code associated with some field or parameter, don't use `@Singular` and add everything you need manually.

The sorted collections (java.util: `SortedSet`, `NavigableSet`, `SortedMap`, `NavigableMap` and guava: `ImmutableSortedSet`, `ImmutableSortedMap`) require that the type argument of the collection has natural order (implements `java.util.Comparable`). There is no way to pass an explicit `Comparator` to use in the builder.

An `ArrayList` is used to store added elements as call methods of a `@Singular` marked field, if the target collection is from the `java.util` package, *even if the collection is a set or map*. Because lombok ensures that generated collections are compacted, a new backing instance of a set or map must be constructed anyway, and storing the data as an `ArrayList` during the build process is more efficient that storing it as a map or set. This behavior is not externally visible, an implementation detail of the current implementation of the `java.util` recipes for `@Singular @Builder`.

With `toBuilder = true` applied to methods, any type parameter of the annotated method itself must also show up in the return type.

The initializer on a `@Builder.Default` field is removed and stored in a static method, in order to guarantee that this initializer won't be executed at all if a value is specified in the build. This does mean the initializer cannot refer to `this`, `super` or any non-static member. If lombok generates a constructor for you, it'll also initialize this field with the initializer.

The generated field in the builder to represent a field with a `@Builder.Default` set is called *`propertyName$value`*; an additional boolean field called *`propertyName$set`* is also generated to track whether it has been set or not. This is an implementation detail; do not write code that interacts with these fields. Instead, invoke the generated builder-setter method if you want to set the property inside a custom method inside the builder.

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

You can suppress the generation of the `builder()` method, for example because you *just* want the `toBuilder()` functionality, by using: `@Builder(builderMethodName = "")`. Any warnings about missing `@Builder.Default` annotations will disappear when you do this, as such warnings are not relevant when only using `toBuilder()` to make builder instances.

You can use `@Builder` for copy constructors: `foo.toBuilder().build()` makes a shallow clone. Consider suppressing the generating of the `builder` method if you just want this functionality, by using: `@Builder(toBuilder = true, builderMethodName = "")`.

Due to a peculiar way javac processes static imports, trying to do a non-star static import of the static `builder()` method won't work. Either use a star static import: `import static TypeThatHasABuilder.*;` or don't statically import the `builder` method.

If setting the access level to `PROTECTED`, all methods generated inside the builder class are actually generated as `public`; the meaning of the `protected` keyword is different inside the inner class, and the precise behavior that `PROTECTED` would indicate (access by any source in the same package is allowed, as well as any subclasses *from the outer class, marked with `@Builder`* is not possible, and marking the inner members `public` is as close as we can get.

If you have configured a nullity annotation flavour via `lombok.config` key `lombok.addNullAnnotations`, any plural-form generated builder methods for `@Singular` marked properties (these plural form methods take a collection of some sort and add all elements) get a nullity annotation on the parameter. You get a non-null one normally, but if you have configured the behavior on `null` being passed in as collection to `IGNORE`, a nullable annotation is generated instead.