@Value

Immutable classes made very easy.

`@Value` was introduced as experimental feature in lombok v0.11.4.

`@Value` no longer implies `@With` since lombok v0.11.8.

`@Value` promoted to the main `lombok` package since lombok v0.12.0.

Overview

`@Value` is the immutable variant of `@Data`; all fields are made `private` and `final` by default, and setters are not generated. The class itself is also made `final` by default, because immutability is not something that can be forced onto a subclass. Like `@Data`, useful `toString()`, `equals()` and `hashCode()` methods are also generated, each field gets a getter method, and a constructor that covers every argument (except `final` fields that are initialized in the field declaration) is also generated.

In practice, `@Value` is shorthand for: `final @ToString @EqualsAndHashCode @AllArgsConstructor @FieldDefaults(makeFinal = true, level = AccessLevel.PRIVATE) @Getter`, except that explicitly including an implementation of any of the relevant methods simply means that part won't be generated and no warning will be emitted. For example, if you write your own `toString`, no error occurs, and lombok will not generate a `toString`. Also, *any* explicit constructor, no matter the arguments list, implies lombok will not generate a constructor. If you do want lombok to generate the all-args constructor, add `@AllArgsConstructor` to the class. Note that if both `@Builder` and `@Value` are on a class, the package private allargs constructor that `@Builder` wants to make 'wins' over the public one that `@Value` wants to make. You can mark any constructor or method with `@lombok.experimental.Tolerate` to hide them from lombok.

It is possible to override the final-by-default and private-by-default behavior using either an explicit access level on a field, or by using the `@NonFinal` or `@PackagePrivate` annotations. `@NonFinal` can also be used on a class to remove the final keyword.
It is possible to override any default behavior for any of the 'parts' that make up `@Value` by explicitly using that annotation.

With Lombok

```
import lombok.AccessLevel;
import lombok.experimental.NonFinal;
import lombok.experimental.Value;
import lombok.experimental.With;
import lombok.ToString;

@Value public class ValueExample {
  String name;
  @With(AccessLevel.PACKAGE) @NonFinal int age;
  double score;
  protected String[] tags;

  @ToString(includeFieldNames=true)
  @Value(staticConstructor="of")
  public static class Exercise<T> {
    String name;
    T value;
  }
}
```

Vanilla Java

```
import java.util.Arrays;

public final class ValueExample {
  private final String name;
  private int age;
  private final double score;
  protected final String[] tags;

  @java.beans.ConstructorProperties({"name", "age", "score", "tags"})
  public ValueExample(String name, int age, double score, String[] tags) {
    this.name = name;
    this.age = age;
    this.score = score;
    this.tags = tags;
  }

  public String getName() {
    return this.name;
  }

  public int getAge() {
    return this.age;
  }

  public double getScore() {
    return this.score;
  }

  public String[] getTags() {
    return this.tags;
  }

  @java.lang.Override
  public boolean equals(Object o) {
    if (o == this) return true;
    if (!(o instanceof ValueExample)) return false;
    final ValueExample other = (ValueExample)o;
    final Object this$name = this.getName();
    final Object other$name = other.getName();
    if (this$name == null ? other$name != null : !this$name.equals(other$name)) return false;
    if (this.getAge() != other.getAge()) return false;
    if (Double.compare(this.getScore(), other.getScore()) != 0) return false;
    if (!Arrays.deepEquals(this.getTags(), other.getTags())) return false;
    return true;
  }

  @java.lang.Override
  public int hashCode() {
    final int PRIME = 59;
    int result = 1;
    final Object $name = this.getName();
    result = result * PRIME + ($name == null ? 43 : $name.hashCode());
    result = result * PRIME + this.getAge();
    final long $score = Double.doubleToLongBits(this.getScore());
    result = result * PRIME + (int)($score >>> 32 ^ $score);
    result = result * PRIME + Arrays.deepHashCode(this.getTags());
    return result;
  }

  @java.lang.Override
  public String toString() {
    return "ValueExample(name=" + getName() + ", age=" + getAge() + ", score=" + getScore() + ", tags=" + A
  }

  ValueExample withAge(int age) {
    return this.age == age ? this : new ValueExample(name, age, score, tags);
  }

  public static final class Exercise<T> {
    private final String name;
    private final T value;

    private Exercise(String name, T value) {
      this.name = name;
      this.value = value;
    }

    public static <T> Exercise<T> of(String name, T value) {
      return new Exercise<T>(name, value);
    }

    public String getName() {
      return this.name;
    }

    public T getValue() {
      return this.value;
    }

    @java.lang.Override
    public boolean equals(Object o) {
      if (o == this) return true;
      if (!(o instanceof ValueExample.Exercise)) return false;
      final Exercise<?> other = (Exercise<?>)o;
      final Object this$name = this.getName();
      final Object other$name = other.getName();
      if (this$name == null ? other$name != null : !this$name.equals(other$name)) return false;
      final Object this$value = this.getValue();
      final Object other$value = other.getValue();
      if (this$value == null ? other$value != null : !this$value.equals(other$value)) return false;
      return true;
    }

    @java.lang.Override
    public int hashCode() {
      final int PRIME = 59;
      int result = 1;
      final Object $name = this.getName();
      result = result * PRIME + ($name == null ? 43 : $name.hashCode());
      final Object $value = this.getValue();
      result = result * PRIME + ($value == null ? 43 : $value.hashCode());
      return result;
    }

    @java.lang.Override
    public String toString() {
      return "ValueExample.Exercise(name=" + getName() + ", value=" + getValue() + ")";
    }
  }
}
```

Supported configuration keys:

`lombok.value.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@Value` as a warning or error if configured.
`lombok.noArgsConstructor.extraPrivate` = [`true` | `false`] (default: false)
If `true`, lombok will generate a private no-args constructor for any `@Value` annotated class, which sets all fields to default values (null / 0 / false).

Small print

Look for the documentation on the 'parts' of `@Value`: `@ToString`, `@EqualsAndHashCode`, `@AllArgsConstructor`, `@FieldDefaults`, and `@Getter`.

For classes with generics, it's useful to have a static method which serves as a constructor, because inference of generic parameters via static methods works in java6 and avoids having to use the diamond operator. While you can force this by applying an explicit `@AllArgsConstructor(staticConstructor="of")` annotation, there's also the `@Value(staticConstructor="of")` feature, which will make the generated all-arguments constructor private, and generates a public static method named `of` which is a wrapper around this private constructor.

Various well known annotations about nullity cause null checks to be inserted and will be copied to the relevant places (such as the method for getters, and the parameter for the constructor and setters). See Getter/Setter documentation's small print for more information.

`@Value` was an experimental feature from v0.11.4 to v0.11.9 (as `@lombok.experimental.Value`). It has since been moved into the core package. The old annotation is still around (and is an alias). It will eventually be removed in a future version, though.

It is not possible to use `@FieldDefaults` to 'undo' the private-by-default and final-by-default aspect of fields in the annotated class. Use `@NonFinal` and `@PackagePrivate` on the fields in the class to override this behaviour.