# @Data

## All together now: A shortcut for `@ToString`, `@EqualsAndHashCode`, `@Getter` on all fields, `@Setter` on all non-final fields, and `@RequiredArgsConstructor`!

## Overview

`@Data` is a convenient shortcut annotation that bundles the features of `@ToString`, `@EqualsAndHashCode`, `@Getter` / `@Setter` and `@RequiredArgsConstructor` together: In other words, `@Data` generates all the boilerplate that is normally associated with simple POJOs (Plain Old Java Objects) and beans: getters for all fields, setters for all non-final fields, and appropriate `toString`, `equals` and `hashCode` implementations that involve the fields of the class, and a constructor that initializes all final fields, as well as all non-final fields with no initializer that have been marked with `@NonNull`, in order to ensure the field is never null.

`@Data` is like having implicit `@Getter`, `@Setter`, `@ToString`, `@EqualsAndHashCode` and `@RequiredArgsConstructor` annotations on the class (except that no constructor will be generated if any explicitly written constructors already exist). However, the parameters of these annotations (such as `callSuper`, `includeFieldNames` and `exclude`) cannot be set with `@Data`. If you need to set non-default values for any of these parameters, just add those annotations explicitly; `@Data` is smart enough to defer to those annotations.

All generated getters and setters will be `public`. To override the access level, annotate the field or class with an explicit `@Setter` and/or `@Getter` annotation. You can also use this annotation (by combining it with `AccessLevel.NONE`) to suppress generating a getter and/or setter altogether.

All fields marked as `transient` will not be considered for `hashCode` and `equals`. All static fields will be skipped entirely (not considered for any of the generated methods, and no setter/getter will be made for them).

If the class already contains a method with the same name and parameter count as any method that would normally be generated, that method is not generated, and no warning or error is emitted. For example, if you already have a method with signature `equals(AnyType param)`, no `equals` method will be generated, even though technically it might be an entirely different method due to having different parameter types. The same rule applies to the constructor (any explicit constructor will prevent `@Data` from generating one), as well as `toString`, `equals`, and all getters and setters. You can mark any constructor or method with `@lombok.experimental.Tolerate` to hide them from lombok.

`@Data` can handle generics parameters for fields just fine. In order to reduce the boilerplate when constructing objects for classes with generics, you can use the `staticConstructor` parameter to generate a private constructor, as well as a static method that returns a new instance. This way, javac will infer the variable name. Thus, by declaring like so: `@Data(staticConstructor="of") class Foo<T> { private T x;}` you can create new instances of `Foo` by writing: `Foo.of(5);` instead of having to write: `new Foo<Integer>(5);`.

## With Lombok

```
import lombok.AccessLevel;
import lombok.Setter;
import lombok.Data;
import lombok.ToString;

@Data public class DataExample {
  private final String name;
  @Setter(AccessLevel.PACKAGE) private int age;
  private double score;
  private String[] tags;

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

## Vanilla Java

```
import java.util.Arrays;

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

  public DataExample(String name) {
    this.name = name;
  }

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

  void setAge(int age) {
    this.age = age;
  }

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

  public void setScore(double score) {
    this.score = score;
  }

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

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

  public void setTags(String[] tags) {
    this.tags = tags;
  }

  @Override public String toString() {
    return "DataExample(" + this.getName() + ", " + this.getAge() + ", " + this.getScore() + ", " + Arrays.
  }

  protected boolean canEqual(Object other) {
    return other instanceof DataExample;
  }

  @Override public boolean equals(Object o) {
    if (o == this) return true;
    if (!(o instanceof DataExample)) return false;
    DataExample other = (DataExample) o;
    if (!other.canEqual((Object)this)) return false;
    if (this.getName() == null ? other.getName() != null : !this.getName().equals(other.getName())) return
    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;
  }

  @Override public int hashCode() {
    final int PRIME = 59;
    int result = 1;
    final long temp1 = Double.doubleToLongBits(this.getScore());
    result = (result*PRIME) + (this.getName() == null ? 43 : this.getName().hashCode());
    result = (result*PRIME) + this.getAge();
    result = (result*PRIME) + (int)(temp1 ^ (temp1 >>> 32));
    result = (result*PRIME) + Arrays.deepHashCode(this.getTags());
    return result;
  }

  public static 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;
    }

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

    protected boolean canEqual(Object other) {
      return other instanceof Exercise;
    }

    @Override public boolean equals(Object o) {
      if (o == this) return true;
      if (!(o instanceof Exercise)) return false;
      Exercise<?> other = (Exercise<?>) o;
      if (!other.canEqual((Object)this)) return false;
      if (this.getName() == null ? other.getValue() != null : !this.getName().equals(other.getName())) retu
      if (this.getValue() == null ? other.getValue() != null : !this.getValue().equals(other.getValue())) r
      return true;
    }

    @Override public int hashCode() {
      final int PRIME = 59;
      int result = 1;
      result = (result*PRIME) + (this.getName() == null ? 43 : this.getName().hashCode());
      result = (result*PRIME) + (this.getValue() == null ? 43 : this.getValue().hashCode());
      return result;
    }
  }
}
```

## Supported configuration keys:

`lombok.data.flagUsage = [ warning | error ]` (default: not set)
Lombok will flag any usage of `@Data` 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 `@Data` annotated class, which sets all fields to default values (null / 0 / false).

## Small print

See the small print of `@ToString`, `@EqualsAndHashCode`, `@Getter / @Setter` and `@RequiredArgsConstructor`.

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.

By default, any variables that start with a $ symbol are excluded automatically. You can include them by specifying an explicit annotation (`@Getter` or `@ToString`, for example) and using the 'of' parameter.