@Getter and @Setter

Never write `public int getFoo() {return foo;}` again.

Overview

You can annotate any field with `@Getter` and/or `@Setter`, to let lombok generate the default getter/setter automatically.
A default getter simply returns the field, and is named `getFoo` if the field is called `foo` (or `isFoo` if the field's type is `boolean`). A default setter is named `setFoo` if the field is called `foo`, returns `void`, and takes 1 parameter of the same type as the field. It simply sets the field to this value.

The generated getter/setter method will be `public` unless you explicitly specify an `AccessLevel`, as shown in the example below. Legal access levels are `PUBLIC`, `PROTECTED`, `PACKAGE`, and `PRIVATE`.

You can also put a `@Getter` and/or `@Setter` annotation on a class. In that case, it's as if you annotate all the non-static fields in that class with the annotation.

You can always manually disable getter/setter generation for any field by using the special `AccessLevel.NONE` access level. This lets you override the behaviour of a `@Getter`, `@Setter` or `@Data` annotation on a class.

To put annotations on the generated method, you can use `onMethod=@__({@AnnotationsHere})`; to put annotations on the only parameter of a generated setter method, you can use `onParam=@__({@AnnotationsHere})`. Be careful though! This is an experimental feature. For more details see the documentation on the onX feature.

NEW in lombok v1.12.0: javadoc on the field will now be copied to generated getters and setters. Normally, all text is copied, and `@return` is *moved* to the getter, whilst `@param` lines are *moved* to the setter. Moved means: Deleted from the field's javadoc. It is also possible to define unique text for each getter/setter. To do that, you create a 'section' named `GETTER` and/or `SETTER`. A section is a line in your javadoc containing 2 or more dashes, then the text 'GETTER' or 'SETTER', followed by 2 or more dashes, and nothing else on the line. If you use sections, `@return` and `@param` stripping for that section is no longer done (move the `@return` or `@param` line into the section).

With Lombok

```java
import lombok.AccessLevel;
import lombok.Getter;
import lombok.Setter;

public class GetterSetterExample {
  /**
   * Age of the person. Water is wet.
   *
   * @param age New value for this person's age. Sky is blue.
   * @return The current value of this person's age. Circles are round.
   */
  @Getter @Setter private int age = 10;

  /**
   * Name of the person.
   * -- SETTER --
   * Changes the name of this person.
   *
   * @param name The new value.
   */
  @Setter(AccessLevel.PROTECTED) private String name;

  @Override public String toString() {
    return String.format("%s (age: %d)", name, age);
  }
}
```

Vanilla Java

```java
public class GetterSetterExample {
  /**
   * Age of the person. Water is wet.
   */
  private int age = 10;

  /**
   * Name of the person.
   */
  private String name;

  @Override public String toString() {
    return String.format("%s (age: %d)", name, age);
  }

  /**
   * Age of the person. Water is wet.
   *
   * @return The current value of this person's age. Circles are round.
   */
  public int getAge() {
    return age;
  }

  /**
   * Age of the person. Water is wet.
   *
   * @param age New value for this person's age. Sky is blue.
   */
  public void setAge(int age) {
    this.age = age;
  }

  /**
   * Changes the name of this person.
   *
   * @param name The new value.
   */
  protected void setName(String name) {
    this.name = name;
  }
}
```

Supported configuration keys:

`lombok.accessors.chain` = [`true` | `false`] (default: false)
If set to `true`, generated setters will return `this` (instead of `void`). An explicitly configured `chain` parameter of an `@Accessors` annotation takes precedence over this setting.
`lombok.accessors.fluent` = [`true` | `false`] (default: false)
If set to `true`, generated getters and setters will not be prefixed with the bean-standard '`get`, `is` or `set`; instead, the methods will use the same name as the field (minus prefixes). An explicitly configured `fluent` parameter of an `@Accessors` annotation takes precedence over this setting.
`lombok.accessors.prefix` += *a field prefix* (default: empty list)
This is a list property; entries can be added with the `+=` operator. Inherited prefixes from parent config files can be removed with the `-=` operator. Lombok will strip any matching field prefix from the name of a field in order to determine the name of the getter/setter to generate. For example, if `m` is one of the prefixes listed in this setting, then a field named `mFoobar` will result in a getter named `getFoobar()`, not `getMFoobar()`. An explicitly configured `prefix` parameter of an `@Accessors` annotation takes precedence over this setting.
`lombok.getter.noIsPrefix` = [`true` | `false`] (default: false)
If set to `true`, getters generated for `boolean` fields will use the `get` prefix instead of the default `is` prefix, and any generated code that calls getters, such as `@ToString`, will also use `get` instead of `is`
`lombok.accessors.capitalization` = [`basic` | `beanspec`] (default: basic)
Controls how tricky cases like `uShaped` (one lowercase letter followed by an upper/titlecase letter) are capitalized. `basic` capitalizes that to `getUShaped`, and `beanspec` capitalizes that to `getuShaped` instead.
Both strategies are commonly used in the java ecosystem, though `beanspec` is more common.
`lombok.setter.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@Setter` as a warning or error if configured.
`lombok.getter.flagUsage` = [`warning` | `error`] (default: not set)
Lombok will flag any usage of `@Getter` 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 setter parameter, and to 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.

Small print

For generating the method names, the first character of the field, if it is a lowercase character, is title-cased, otherwise, it is left unmodified. Then, get/set/is is prefixed.

No method is generated if any method already exists with the same name (case insensitive) and same parameter count. For example, `getFoo()` will not be generated if there's already a method `getFoo(String... x)` even though it is technically possible to make the method. This caveat exists to prevent confusion. If the generation of a method is skipped for this reason, a warning is emitted instead. Varargs count as 0 to N parameters. You can mark any method with `@lombok.experimental.Tolerate` to hide them from lombok.

For `boolean` fields that start with `is` immediately followed by a title-case letter, nothing is prefixed to generate the getter name.

Any variation on `boolean` will *not* result in using the `is` prefix instead of the `get` prefix; for example, returning `java.lang.Boolean` results in a `get` prefix, not an `is` prefix.

A number of annotations from popular libraries that indicate non-nullness, such as `javax.annotation.Nonnull`, if present on the field, result in an explicit null check in the generated setter.

Various well-known annotations about nullability, such as `org.eclipse.jdt.annotation.NonNull`, are automatically copied over to the right place (method for getters, parameter for setters). You can specify additional annotations that should always be copied via lombok configuration key `lombok.copyableAnnotations`.

You can annotate a class with a `@Getter` or `@Setter` annotation. Doing so is equivalent to annotating all non-static fields in that class with that annotation. `@Getter` / `@Setter` annotations on fields take precedence over the ones on classes.

Using the `AccessLevel.NONE` access level simply generates nothing. It's useful only in combination with `@Data` or a class-wide `@Getter` or `@Setter`.

`@Getter` can also be used on enums. `@Setter` can't, not for a technical reason, but for a pragmatic one: Setters on enums are an extremely bad idea.