[Difference between @Mock and @InjectMocks](https://stackoverflow.com/questions/16467685/difference-between-mock-and-injectmocks)

@Mock creates a mock. @InjectMocks creates an instance of the class and injects the mocks that are created with the @Mock (or @Spy) annotations into this instance.

Note that you must use @RunWith(MockitoJUnitRunner.class) or Mockito.initMocks(this) to initialize these mocks and inject them.

This is a sample code on how @Mock and @InjectMocks works.

Say we have Game and Player class.

class Game {

private Player player;

public Game(Player player) {

this.player = player;

}

public String attack() {

return "Player attack with: " + player.getWeapon();

}

}

class Player {

private String weapon;

public Player(String weapon) {

this.weapon = weapon;

}

String getWeapon() {

return weapon;

}

}

As you see, Game class need Player to perform an attack.

@RunWith(MockitoJUnitRunner.class)

class GameTest {

@Mock

Player player;

@InjectMocks

Game game;

@Test

public void attackWithSwordTest() throws Exception {

Mockito.when(player.getWeapon()).thenReturn("Sword");

assertEquals("Player attack with: Sword", game.attack());

}

}

Mockito will mock a Player class and it's behaviour using when and thenReturn method. Lastly, using @InjectMocks Mockito will put that Player into Game.

Notice that you don't even have to create a new Game object. Mockito will inject it for you.

// you don't have to do this

Game game = new Game(player);

We will also get same behaviour using @Spy annotation. Even if the attribute name is different.

@RunWith(MockitoJUnitRunner.class)

public class GameTest {

@Mock Player player;

@Spy List<String> enemies = new ArrayList<>();

@InjectMocks Game game;

@Test public void attackWithSwordTest() throws Exception {

Mockito.when(player.getWeapon()).thenReturn("Sword");

enemies.add("Dragon");

enemies.add("Orc");

assertEquals(2, game.numberOfEnemies());

assertEquals("Player attack with: Sword", game.attack());

}

}

class Game {

private Player player;

private List<String> opponents;

public Game(Player player, List<String> opponents) {

this.player = player;

this.opponents = opponents;

}

public int numberOfEnemies() {

return opponents.size();

}

// ...

That's because Mockito will check the Type Signature of Game class, which is Player and List<String>.

**Mockito: Why You Should Not Use InjectMocks Annotation to Autowire Fields**

[Ted Vinke](https://tedvinke.wordpress.com/author/tedvinke/) [Java](https://tedvinke.wordpress.com/category/java/)  13/02/2014 3 Minutes

People like the way how Mockito is able to mock Spring’s auto-wired fields with the @InjectMocks annotation. When I read [this post of Lubos Krnac last week](http://lkrnac.net/blog/2014/01/21/mock-autowired-fields/), I thought I should explain why I think the use of InjectMocks is a bad signal and how you should avoid it. Hint: it’s about visibility.

Let’s say we have a PlannerServiceImpl which delegates to a PlannerClient. Uses Spring for auto-wiring all together; there’s no constructor, but Spring is able to use *field injection*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @Service  public class PlannerServiceImpl implements PlannerService {      private static final Logger LOG = LoggerFactory.getLogger(PlannerServiceImpl.class);        @Autowired      private PlannerClient plannerClient;        @Override      public Long createWeddingPlan() {          try {              CreateWeddingPlanResponse response = plannerClient.createWeddingPlan();              return convert(response).getId();          } catch (Exception e) {              LOG.error("Unable to create wedding plan", e);              return null;          }      } |

An associated test could look like:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @RunWith(MockitoJUnitRunner.class)  public class PlannerServiceImplTest {        @Mock      private PlannerClient plannerClient;        @InjectMocks      private final PlannerServiceImpl plannerService = new PlannerServiceImpl();        @Test      public void testCreateWeddingPlanWhenClientReturnsUndefinedResponseThenNullIsReturned() throws Exception {          when(plannerClient.createWeddingPlan()).thenReturn(null);            final Long actual = plannerService.createWeddingPlan();            assertThat(actual, is(nullValue()));      } |

The org.mockito.InjectMocks annotation can be seen as an equivalent of Spring’s own dependency injection. The Javadoc states:

*Mockito will try to inject mocks only either by constructor injection, setter injection, or property injection in order and as described below. If any of the following strategy fail, then Mockito****won’t report failure****; i.e. you will have to provide dependencies yourself.*

(Whoever would design this to fail silently at all?)

So what if someone decides to **create a new dependency**, say an AuditService and upgrades a bunch of services by adding it as an additional property, also marked as @Autowired?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @Service  public class PlannerServiceImpl implements PlannerService {      private static final Logger LOG = LoggerFactory.getLogger(PlannerServiceImpl.class);        @Autowired      private PlannerClient plannerClient;        @Autowired      private AuditService auditService;        @Override      public Long createWeddingPlan() {          try {              CreateWeddingPlanResponse response = plannerClient.createWeddingPlan();              auditService.addEntry("Wedding plan created.");              return convert(response).getId();          } |

The test will fail, probably on a NullPointerException on a missing AuditService – and **it is not visible why**. InjectMocks will fail silently and there’s no indication the test needs this. *Did I already ask whoever would design something like this to fail****silently****?*

If you’re doing TDD or not (and we are able to change the test first) – clients of this code don’t know about an additional dependency, because it’s completely hidden. You shouldn’t use InjectMocks to deal with injecting private fields (err..or at all) , because this kind of Dependency Injection is **evil** – and signals you should change your design.

There, I said it.

**Fix #1: Solve your design and make your dependencies visible.**

Create a constructor. Pass along the PlannerClient.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Service  public class PlannerServiceImpl implements PlannerService {      private static final Logger LOG = LoggerFactory.getLogger(PlannerServiceImpl.class);        private final PlannerClient plannerClient;        @Autowired      public PlannerServiceImpl(final PlannerClient plannerClient) {          this.plannerClient = plannerClient;      } |

Now, when there are more dependencies needed, they’re clearly in sight [because the constructor says so](http://blog.schauderhaft.de/2012/01/01/the-one-correct-way-to-do-dependency-injection/). So don’t go creating a bunch of setters now – they still don’t force you to pass along your required dependencies!

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @Service  public class PlannerServiceImpl implements PlannerService {      private static final Logger LOG = LoggerFactory.getLogger(PlannerServiceImpl.class);        private final PlannerClient plannerClient;        private final AuditService auditService;        @Autowired      PlannerServiceImpl(PlannerClient plannerClient, AuditService auditService) {          this.plannerClient = plannerClient;          this.auditService = auditService;      } |

The test itself won’t compile any more (luckily, because of the way we’ve been instantiating the field as plannerService = new PlannerServiceImpl()!) as soon as e.g. the AuditService is added to the constructor. So it’s time to..

**Fix #2: Get rid of @InjectMocks**

There’s no need to use @InjectMocks anymore. Instead instantiate the class-under-test properly in a @Before-annotated method – where it belongs, passing along all needed dependencies.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @RunWith(MockitoJUnitRunner.class)  public class PlannerServiceImplTest {    @Mock  private PlannerClient plannerClient;    @Mock  private AuditService auditService;    private PlannerServiceImpl plannerService;    @Before  void setUp() {      plannerService = new PlannerServiceImpl(plannerClient, auditService);  } |

* **Tagged**
* [Mockito](https://tedvinke.wordpress.com/tag/mockito/)

1. https://0.gravatar.com/avatar/367fb39afa7959e2b6c9e4db4845cfa4?s=48&d=https%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D48&r=G**Darrell Burgan**

[16/02/2014 at 02:09](https://tedvinke.wordpress.com/2014/02/13/mockito-why-you-should-not-use-injectmocks-annotation-to-autowire-fields/#comment-171)

You have a good point, but I disagree with you for two reasons.

The first is that the intent of TDD is to do white box testing, not just black box testing. So it is permissible, in my view, for a unit test to know more about what is going on than what is visible via the public facade of a class. Note I said “permissible”, not “desirable”.

The second reason is bigger. While it is a good thing to have the compiler tell you when you’re missing a dependency, you are essentially advocating that people should not use the Spring or JEE dependency injection annotations, which is a far bigger topic than just how do you do unit testing using Mockito. Essentially you are arguing against that paradigm of development in favor of explicit dependency injection via constructors. That may be a superior programming approach (or it may not) but either way it is a big change from common practice. I’d love to hear your thoughts on that.

Cheers,  
Darrell

Liked by [1 person](https://tedvinke.wordpress.com/2014/02/13/mockito-why-you-should-not-use-injectmocks-annotation-to-autowire-fields/)

1. https://1.gravatar.com/avatar/1cd7b5629508f84d86a752c9a346f7ae?s=48&d=https%3A%2F%2F1.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D48&r=G[**David Grant**](http://gravatar.com/davidgrant)

[17/02/2014 at 07:19](https://tedvinke.wordpress.com/2014/02/13/mockito-why-you-should-not-use-injectmocks-annotation-to-autowire-fields/#comment-172)

IMHO, setter injection should be banished forever, and should be replaced with constructor injection. I have never had a situation where I couldn’t use constructor injection instead of setter injection. My experience includes 2 large projects using Guice and 1 large project using Spring (which we migrated from setter injection to constructor injection).

Darrell, when Spring started I think setter injection was strongly favoured. I’m not sure why, but anyways, that paradigm became very entrenched. Slowly, constructor injection is becoming more popular.

Check out this blog post:  
<http://spring.io/blog/2007/07/11/setter-injection-versus-constructor-injection-and-the-use-of-required>  
where they talk about the history of setter vs constructor injection and give a recommendation (use constructor injection).

Like

* 1. https://0.gravatar.com/avatar/367fb39afa7959e2b6c9e4db4845cfa4?s=48&d=https%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D48&r=G**Darrell Burgan**

[18/02/2014 at 05:27](https://tedvinke.wordpress.com/2014/02/13/mockito-why-you-should-not-use-injectmocks-annotation-to-autowire-fields/#comment-173)

Good blog post. Like I said, having the compiler track the dependencies is superior than runtime injection usually. But there are still cases to be made for deferring the binding until the last moment, along with the many cases against it. Food for thought ….

Liked by [1 person](https://tedvinke.wordpress.com/2014/02/13/mockito-why-you-should-not-use-injectmocks-annotation-to-autowire-fields/)

1. https://0.gravatar.com/avatar/0162e4fc0714206d466b196acc7cb3ea?s=48&d=https%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D48&r=G[**Ted Vinke**](https://tedvinke.wordpress.com/)

[18/02/2014 at 16:39](https://tedvinke.wordpress.com/2014/02/13/mockito-why-you-should-not-use-injectmocks-annotation-to-autowire-fields/#comment-174)

Thanks David, you beat me to it.

Darrell, I’m **not** advocating that people should not use the Spring or JEE dependency injection annotations, because I’ve added a constructor, still to be used with Spring’s auto-wiring right? But it’s a constructor (instead of field or setter injection) and that’s the proper way of doing it IMHO.

Consequently, in the test itself *no* @InjectMocks magic is needed anymore.

Both thanks for your responses!