به نام خدا

**دوره کیفیت نرم افزار**

موضوع جلسه : آزمون واحد (Unit Testing)

مهدی نیک نژاد

سرفصل ها :

* Test
* Unit Test
* Junit
* Exercise

فهرست مطالب

[**اهمیت آزمون واحد :** 2](#_Toc47001261)

[**تست نرم افزار :** 2](#_Toc47001262)

[**ابعاد آزمون نرم افزار :** 2](#_Toc47001263)

[**معنای آزمون واحد :** 3](#_Toc47001264)

[**روش سنتی آزمون واحد :** 4](#_Toc47001265)

[**ویژگی های آزمون واحد :** 5](#_Toc47001266)

[**آشنایی با Junit :** 5](#_Toc47001267)

[**اجرای آزمون واحد :** 7](#_Toc47001268)

[**نمونه آزمون یا test case :** 8](#_Toc47001269)

[**مجموعه Junit Assertions :** 8](#_Toc47001270)

[**سایر امکانات Junit :** 9](#_Toc47001271)

[**مزایای آزمون واحد :** 12](#_Toc47001272)

[**زمان تولید آزمون واحد :** 12](#_Toc47001273)

[**کیفیت آزمون های واحد :** 12](#_Toc47001274)

[**ویژگی آزمون های خوب :** 13](#_Toc47001275)

[**برنامه نویسی مبتنی بر تست (TDD) :** 13](#_Toc47001276)

[**بررسی چند مثال :** 15](#_Toc47001277)

[**جمع بندی :** 19](#_Toc47001278)

# **اهمیت آزمون واحد :**

نرم افزار هم یک محصول است و باید قبل از ارایه به مشتری تست شود . پس کنترل کیفیت و تضمین کیفیت در صنایع مختلف از جمله حوزه نرم افزار مهم هستند . پس نرم افزاری که هنوز تست نشده است هنوز تکنمیل نشده است .

**ویژگی های نرم افزار خوب :**

* عملکرد صحیح : منطبق با طراحی و نیازمندی های کاربرFunctional Requirements
* ویژگی های کیفی(غیر عملکردی) : کارایی ، استفاده بهینه از منابع ، سرعت مناسب و امنیت و .... nonfunctional Requirements

# **تست نرم افزار :**

فرآیندی که در طی آن ویژگی های نرم افزار را تست میکنیم .یعنی مثلا چندین نوع تست داریم :

* تست کارایی
* تست سرعت
* تست امنیت
* .....

# **ابعاد آزمون نرم افزار :**

* سطح آزمون : در چه حدی نرم افزار را تست میکنیم ؟

طبق نمودار v-Model داریم :

* + Acceptance Test : تست کردن نرم افزار بعد از deploy مطابق با سناریو های کاربر
  + System Test : تست کردن کل نرم افزار
  + Integration Test : تست کردن چند بخش یکپارچه / ببینیم چند بخش در کنار هم کار میکنند ؟
  + Unit Test : تست کردن یک مولفه کوچک به طور مستقل



* نوع آزمون : functional یا non-functional ؟
* روش آزمون :
  + white box : در تست کردن جزییات پیاده سازی را هم در نظر بگیریم
  + black box : در این روش تست ، به پیاده سازی کاری نداریم و فقط به ورودی و خروجی کار داریم و آنها را فقط تست میکنیم .
* شکل آزمون : دستی یا خودکار manual or automatic ؟
  + آزمونگر کیست ؟ خود برنامه نویس یا کاربر یا ...

در اینجا فقط unit test را بررسی میکنیم .

انواع مختلف تستی که گفتیم در طول عمر یک پروژه انجام میشوند .برخی با دخالت مشتری (stackhlder) و برخی توسط تیم QC یا کنترل کیفیت . اما آزمون واحد یا unit Test توسط خود برنامه نویس انجام مشود و وظیفه او میباشد .

# **معنای آزمون واحد :**

تست کردن بخش کوچکی از یک برنامه مثلا یک متد یا یک کلاس مستقل از وابستگی های آن

حالا باید بررسی کنیم که عملکردش صحیح هست یا خیر . یعنی به ازای ورودی های مختلف ، خروجی های مناسب تولید میکند یا خیر ؟

تاکید میکنیم که هر بخش را جداگانه تست میکنیم .

مثال : تست کردن هر قطعه خودرو قبل از اسمبل کردن کل آن

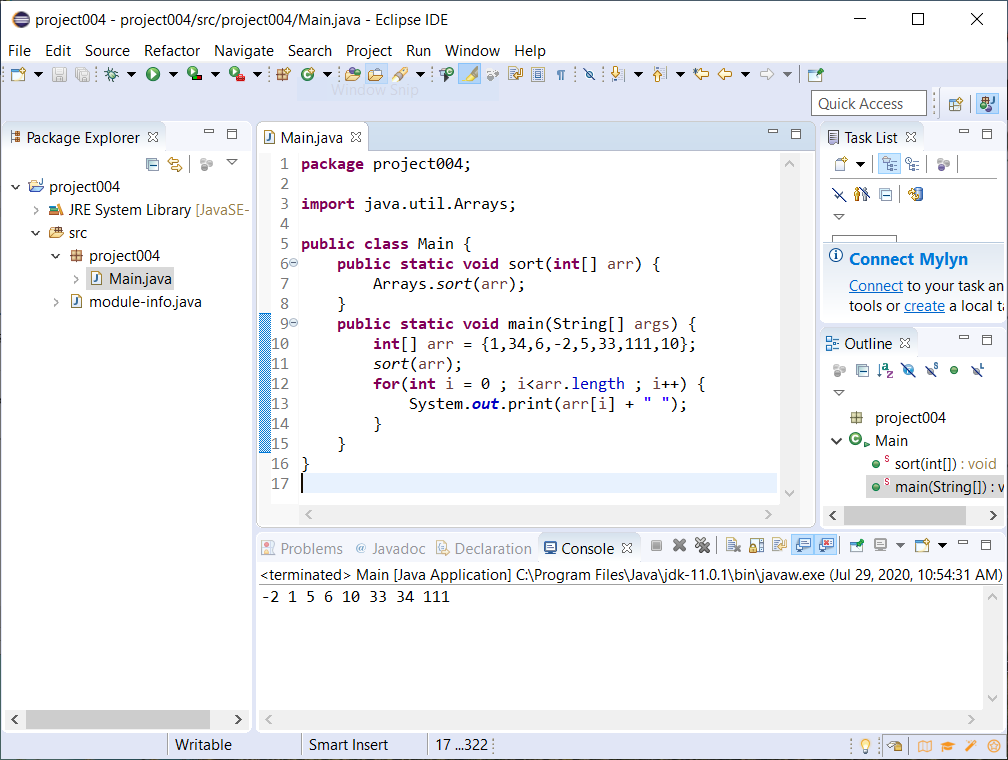
# **روش سنتی آزمون واحد :**

فرض : میخواهیم یک متد sort که نوشتیم را تست کنیم . خب در این روش سنتی یک متد main مینویسیم . و چند حالت محتلف را به عنوان ورودی به ایت متد میدهیم و بعد نتایج را چاپ میکنیم و آنها را به صورت چشمی و دستی بررسی میکنیم و بعد کد های تست شده را comment میکنیم . و سراغ ادامه کار میرویم .

به متن اصلی برنامه Business code میگویند مثلا متد sort

و کد هایی که برای آزمایش نوشته میشوند را Test code میگویند .

مثال :



اما اشکال کجاست؟

* تست ها را داریم دور میریزیم در حالیکه میشود از آنها استفاده مجدد reuse کرد .
* در هر لحظه دارد یک تست فقط انجام میشود .
* برنامه نویس به صورت دستی دارد تست ها را اجرا میکند .
* برنامه نویس باید شخصا تست ها را بررسی کند و اتوماتیک نیست .

پس :

# **ویژگی های آزمون واحد :**

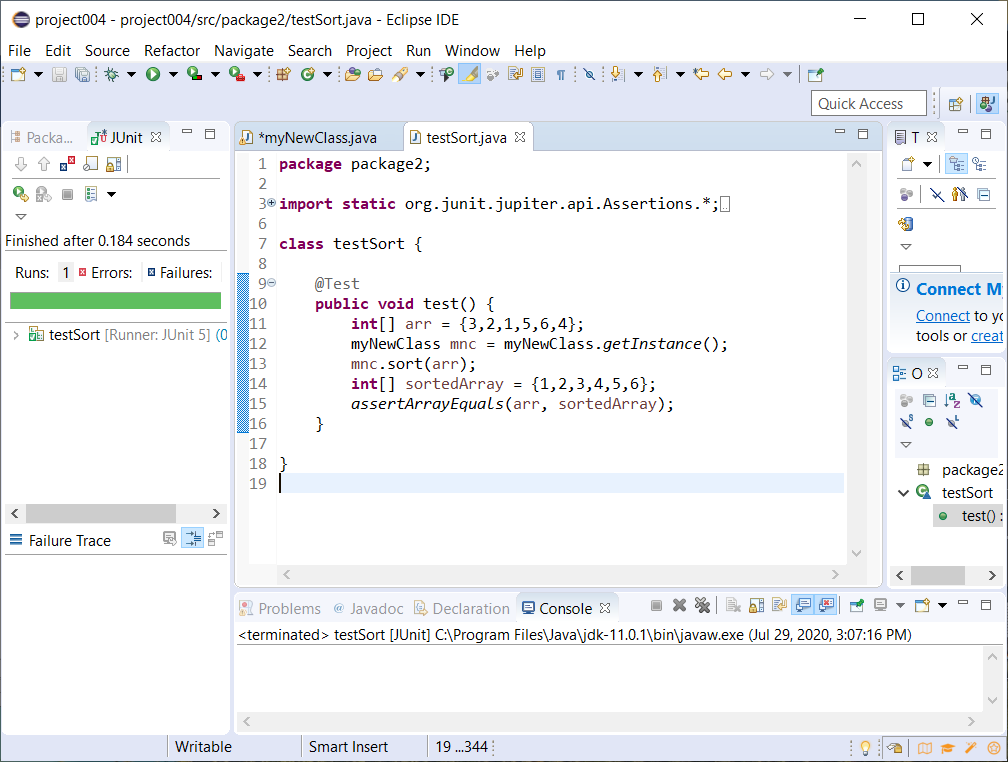
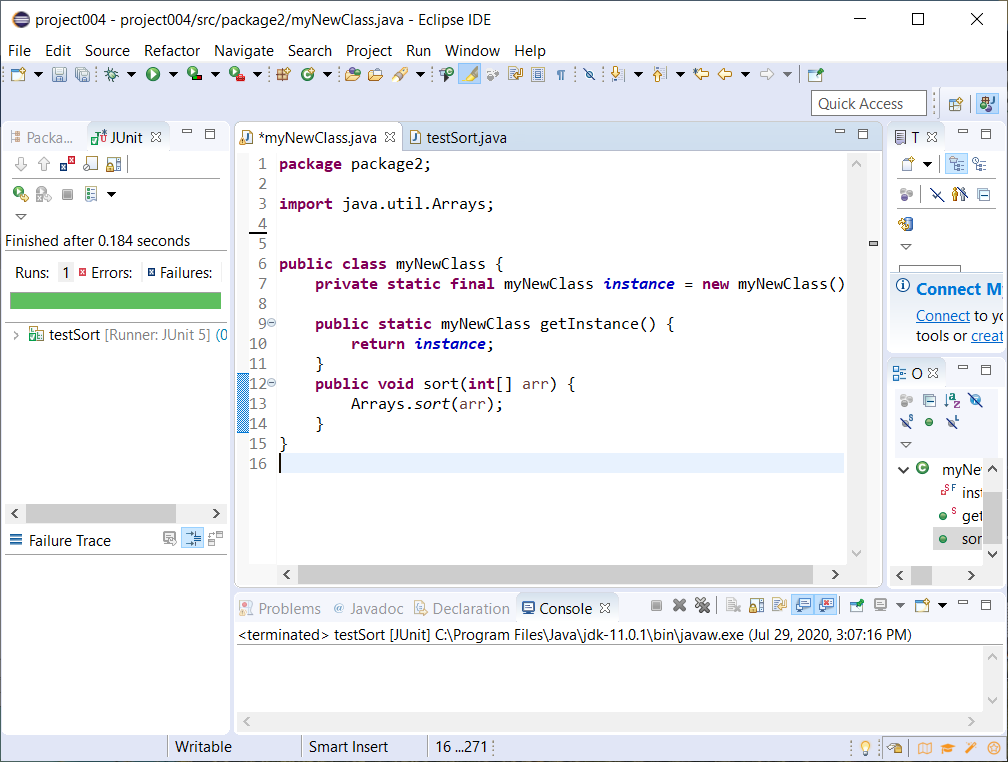
* اجرای خودکار
* مشخص شدن نتیجه به صورت خودکار
* قابل تکرار reuse
* روش آزمون : white box . چون وارد جزییات پیاده سازی میشویم .
* نوع آزمون : معمولا صحت عملکرد را تست میکنیم و functional هست .
* توسط خود برنامه نویس هم باید حتما تست شود .

فریمورک هایی برای آزمون واحد داریم که نام مجموعه کلی آنها XUnit هست . برای مثال :

* cppUnit برای زبان cpp
* Junit برای زبان Java

# **آشنایی با Junit :**

اولین نمونه تست : (test case)



در اینجا به 2 مورد برخورد کردیم :

* annotation @Test
* assertArrayEquals method : برابری 2 تا آرایه پاس داده شده به آنرا جک میکند .

بعد از این اجرای تست ، یک نوار سبز رنگ پدید می آید .

# **اجرای آزمون واحد :**

* test case ها متد main ندارند و به کمک Test Runner ها اجرا میشوند .
* تست رانر ها از طریق خود IDE ها و یا Maven اجرا میشوند .
* هر متد تست ، توسط annotation @Test مشخص میشود .
* به هر متد تست ، یک test case میگویند .
* موفقیت آمیز بودن یا نبودن تست باید به صورت خودکار با استفاده از assert ها (ادعا ها ) انجام شود . نه به صورت چاپ کردن با println

# **نمونه آزمون یا test case :**

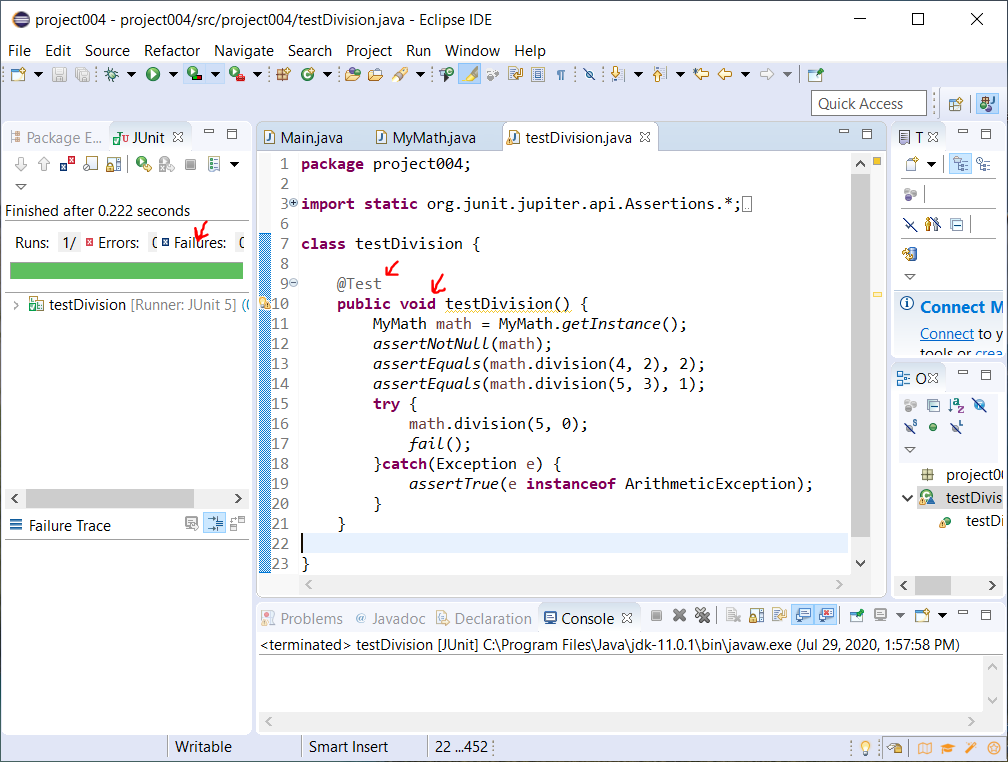
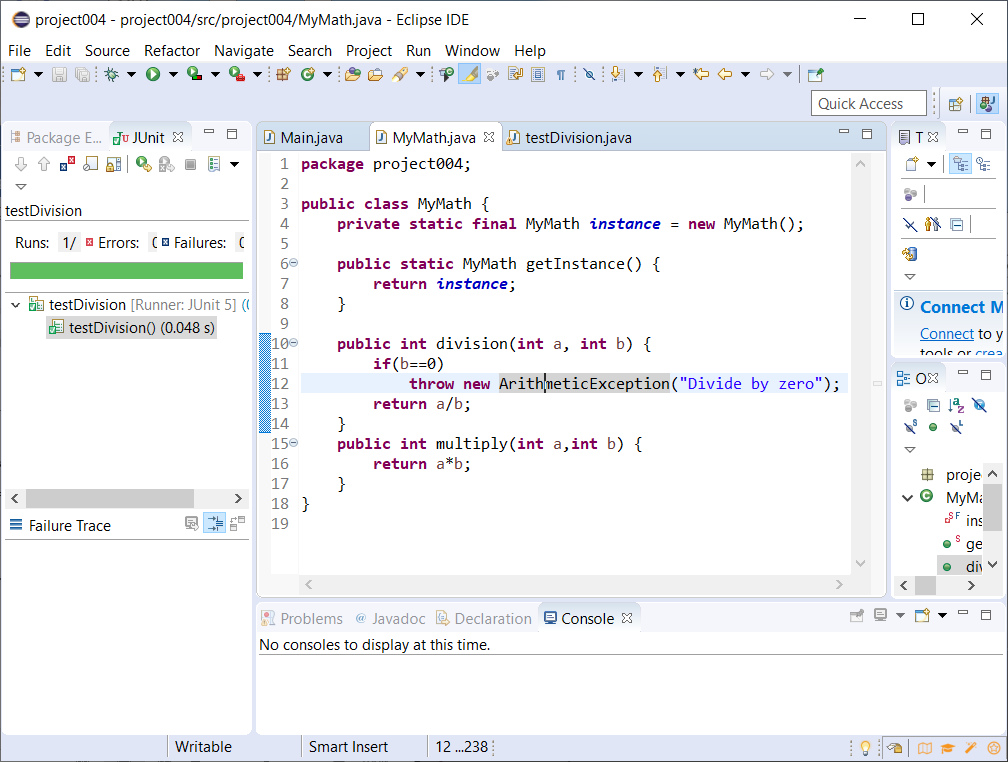
اجزای یک test case :

* یک ورودی test data مثل همان آرایه ای که مشخص کردیم .
* یک خروجی expected result (خروجی مورد انتظار) مثل همان آرایه دیگری مشخص کردیم .
* فراخوانی متد تست با ورودی
* تطبیق خروجی مورد انتظار با خروجی تولید شده توسط متد تست با استفاده از assertion ها
* اگر اجرای تست موفقیت آمیز باشد ، میگوییم که تست pass شده است و اگرنه میگوییم که تست fail شده است .

# **مجموعه Junit Assertions :**

* assertNull(x) : چک میکند که x حتما مقدار Null داشته باشد
* assertNotNull(x) : برعکس بالایی
* assertTrue(Boolean x) : چک میکند که پارامتر x ، True باشد .
* assertFalse(x) : برعکس بالایی
* assertEquals(x,y) : چک میکند که x و y با هم برابر باشند . به معنای x.equals(y)
* assertSame(x,y) : مثل بالایی برابری را چک میکند اما به معنای : x == y
* fail() : اگر اجرای تست به این متد برسد ، آن تست کیس fail میشود . در واقع وقتی از این استفاده میکنیم که نمیخواهیم برنامه به اینجا برسد .

مثال :



توجه : حتما “void” را بگذارید چرا که مهم بود و در غیر آن صورت کد تست اجرا نمیشد . در واقع باید متد باشد نه کانستراکتر .

اولین پارامتر در assertEquals ، همانی است که میخواهیم تست کنیم (expected)و دومی هم مقداری است که بدست می آید (actual) .

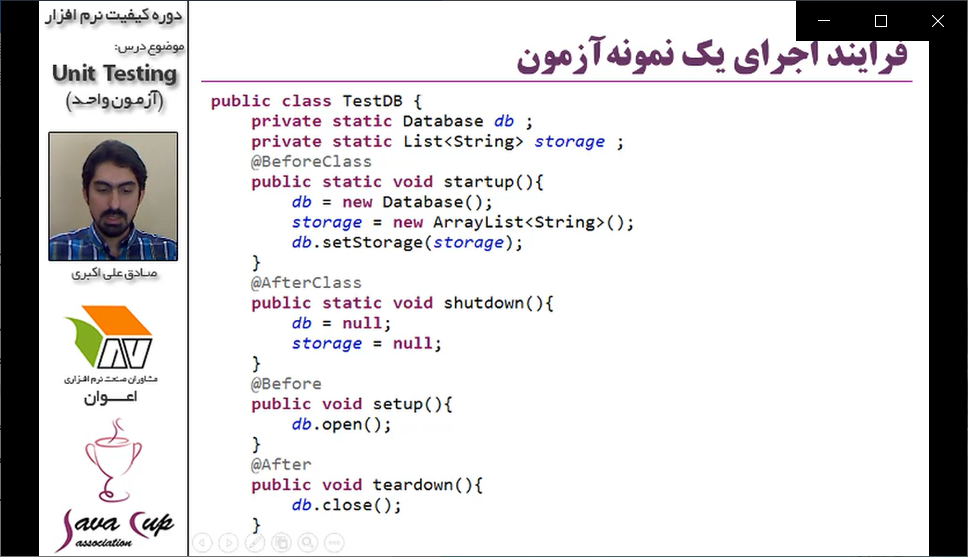
توجه : به math.division(5,0) توجه کنید . توقع داریم که این متد یک Exception پرتاب کند و یعنی نباید به هط بعدی اش برسیم لذا fail() را گذاشتیم . در داخل catch هم Exception را گذاشتیم . پس انتظار داشتیم که اینجا Exception ای که پرتاب میشود از جنس(instanceof) Arethmetic باشد .

# **سایر امکانات Junit :**

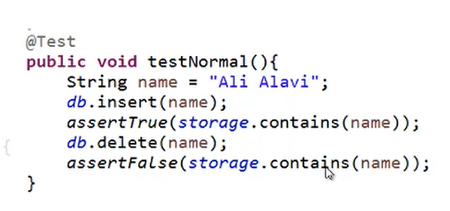
* متد setup که با **@Before** مشخص میشود :
  + قبل از هر متد تست اجرا میشود
  + مسییول کارهایی قبل از اجرای تست . مثلا : اتصال به دیتابیس ، باز کردن فایل ، مقداردهی به فیلدها و ...
  + توجه : @BeforeClass هم داریم که کلا 1 بار قبل از همه تست ها اجرا میشود . در صورتی که @Before قبل از هر تستی اجرا میشود .
* متد teardown که با **@After** مشخص میشود :
  + بعد از هر متد تست اجرا میشود
  + مناسب کارهایی مثل : بستن فایل ، بستن دیتابیس و ....
  + توجه کنید که @AfterClass هم داریم که 1 بار بعد از همه تست ها اجرا میشود .

مثال دیتابیس :

* محل ذخیره همان storage است که اطلاعات در آن ذخیره میشود .
* setStorage() هم برای ذخیره است .
* یکبار برای همه تست ها قبل از اجرا فضا ها را میگیریم با @BeforeClass
* و یکبار هم برای همه تست ها بعد از اجرا فضا ها را پس میدهیم با @AfterClass
* قبل از هر تستی دیتابیس را باز میکنیم با @Before
* و بعد از هر تستی هم دیتابیس را میبندیم با @After



یک تست کیس برای این :

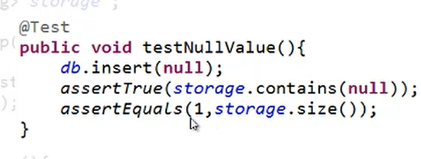


یک رکورد را وارد کردیم و بعدش چک میکنیم که در Storage ذخیره شده باشد و بعد از Delete کردن آن ، چک میکنیم که حتما False برگرداند چون دیگر نباید باشد در Storage .

یک تست کیس دیگر :

در واقع در اینجا قبلش چیزی وارد نشده بوده در دیتابیس . چون گفته که سایز استوریج 1 است !

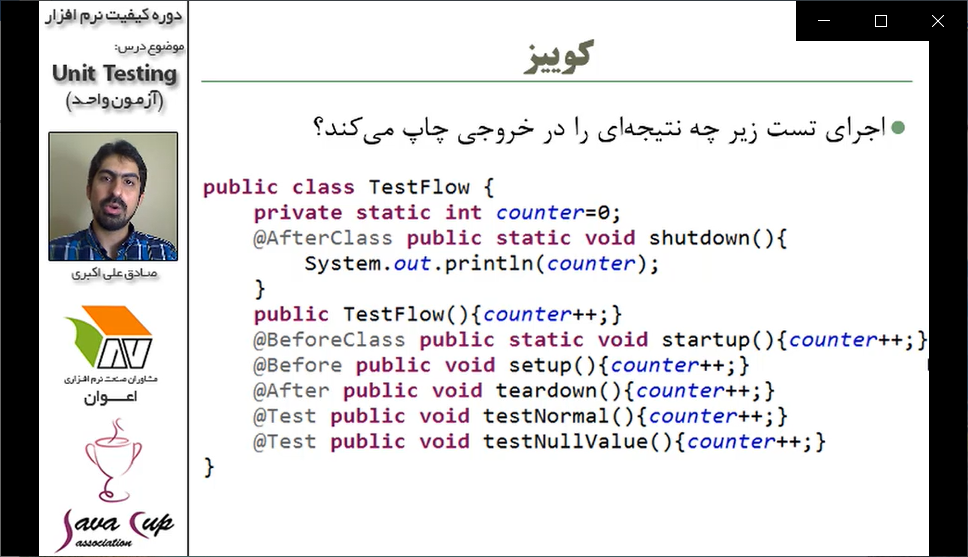
البته این فرض نیامده و لذا این تست کیس زیاد خوب نیست !



روند اجرا :

***BeforeClass -> Before -> After -> Before -> After -> …. -> Before -> After -> AfterClass***

کوییز :



خروجی بعد از اجرای تست های testNormal و testNullValue : ؟

نتیجه ای چاپ نمیشود و تست ها pass میشوند ؟

# **مزایای آزمون واحد :**

* بهبود کیفیت برنامه ها - به دلیل تست شدن برنامه ها
* کاهش bug ها **-** و درنتیجه وقت کمتری به خاطر دیباگ کردن به هدر میرود
* بهبود ساختار و طراحی برنامه – در حین تست کردن متوجه اشکالات business code میشویم و طراحی آنرا تغییر میدهیم .
* افزایش اطمینان نسبت به واحد های کوچک ( سطوح پایین کد )
* مستندات گویا و قابل اجرا
  + نحوه استفاده از business code را مشخص میکند
  + رفتار های مختلف business code را توصیف میکند
  + برتری این مستند نسبت به Javadoc ها این است که :

قطعا مستند صحیحی هست

سازگار باقی میماند با توجه به تغییرات در طی برنامه

# **زمان تولید آزمون واحد :**

* قبل یا بلافاصله بعد از نوشتن واحد
* عواقب به تاخیر انداختن نوشتن Unit Test :
  + کاهش تمرکز روی برنامه ای که نوشتیم و باید برایش تست مینوشتیم
  + کاهش بهبود کیفیت کد
  + به وجود آمدن خسارت ناخواسته (Collateral Damage)
    - یک بخش را اصلاح میکنیم اما باعث اشکال در بخش دیگر میشویم . (می آییم ابروشو درست کنیم میزنیم چشمش رو هم کور میکنیم 😐 )
* برخی تصورات غلط :
  + وقت نداریم لذا تست نمینویسیم !
  + تست نوشتن وظیفه من نیست !

اما

زمانی که صرف نوشتن تست میشود ، هدر نمیرود و به کمک ما می آید در موقع دیباگ کردنها .

# **کیفیت آزمون های واحد :**

خودکار بودن آزمون واحد :

* تمام test case ها باید 100 درصدشان پاس شوند .
* اگر یک تست کیس ، fail شد ، باید فرآیند گسترش کد متوقف شود تا آن اشکال رفع شود .

قواعد تولید آزمون واحد :

* نباید آزمون های بدیهی بنویسیم مثلا برای getter و setter ها
* ورودی های مختلف را باید تست کنیم :
  + ورودی های معمولی
  + ورودی های مرزی مثلا 0
  + ورودی های خاص مثلا null و منفی و آرایه خالی و ...
* برای بخش های حساس تر بیزینس کد ، باید کیفیت آزمون را بالا ببریم (test coverage)

اگر بعد از تعداد مناسبی تست ، باز هم اشکال پیدا شد ، یعنی test case ها کامل نبودند زیرا اگر بودند این اشکالات باید توسط همان تست ها پیدا میشدند .

حالا فرآیند رفع اشکال :

* تست کیسی بنویسیم که این اشکال در آن fail شود
* business code مربوطه اصلاح شود
* آزمون واحد مجدد اجرا شود

# **ویژگی آزمون های خوب :**

برنامه های تست مثل برنامه های اصلی مهم هستند Test code is real code

* خودکار
* کامل : همه سناریو هایی که منجر به خطای احتمالی میشود را باید تست کند و pass شود
* قابل تکرار
* مستقل : کد های تست نباید به چیزی وابسته باشند تا قابل تکرار بودنشان سلب شود و همچنین باید مستقل باشند چون داریم unit را تست میکنیم .
* حرفه ای باشند

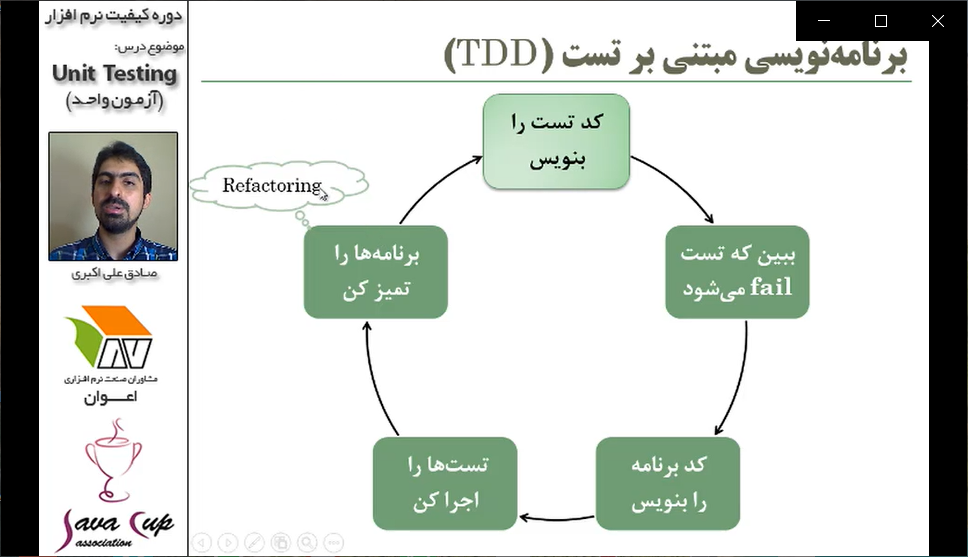
# **برنامه نویسی مبتنی بر تست (TDD) :**

نام های دیگر این رویکرد :

* Test Driven Development
* Test First Development

در این رویکرد ، تست ها قبل از نوشتن کد اصلی (business code) نوشته میشوند.

درابتدا 100 درصد تست ها fail میشوند چون هنوز کدی ننوشتیم تا تست روی آن انجام شود .





فرق TDD با تست سنتی :

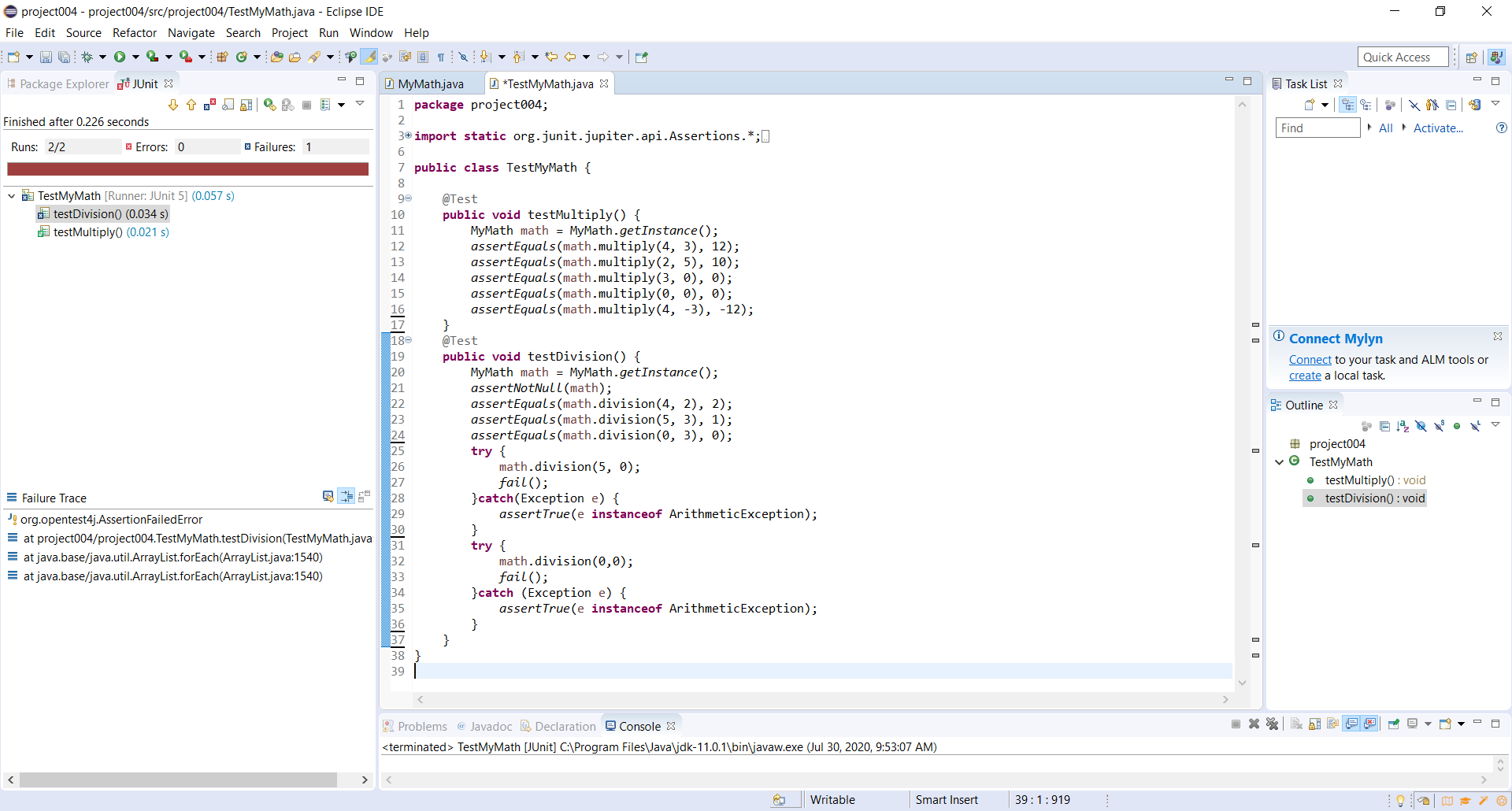
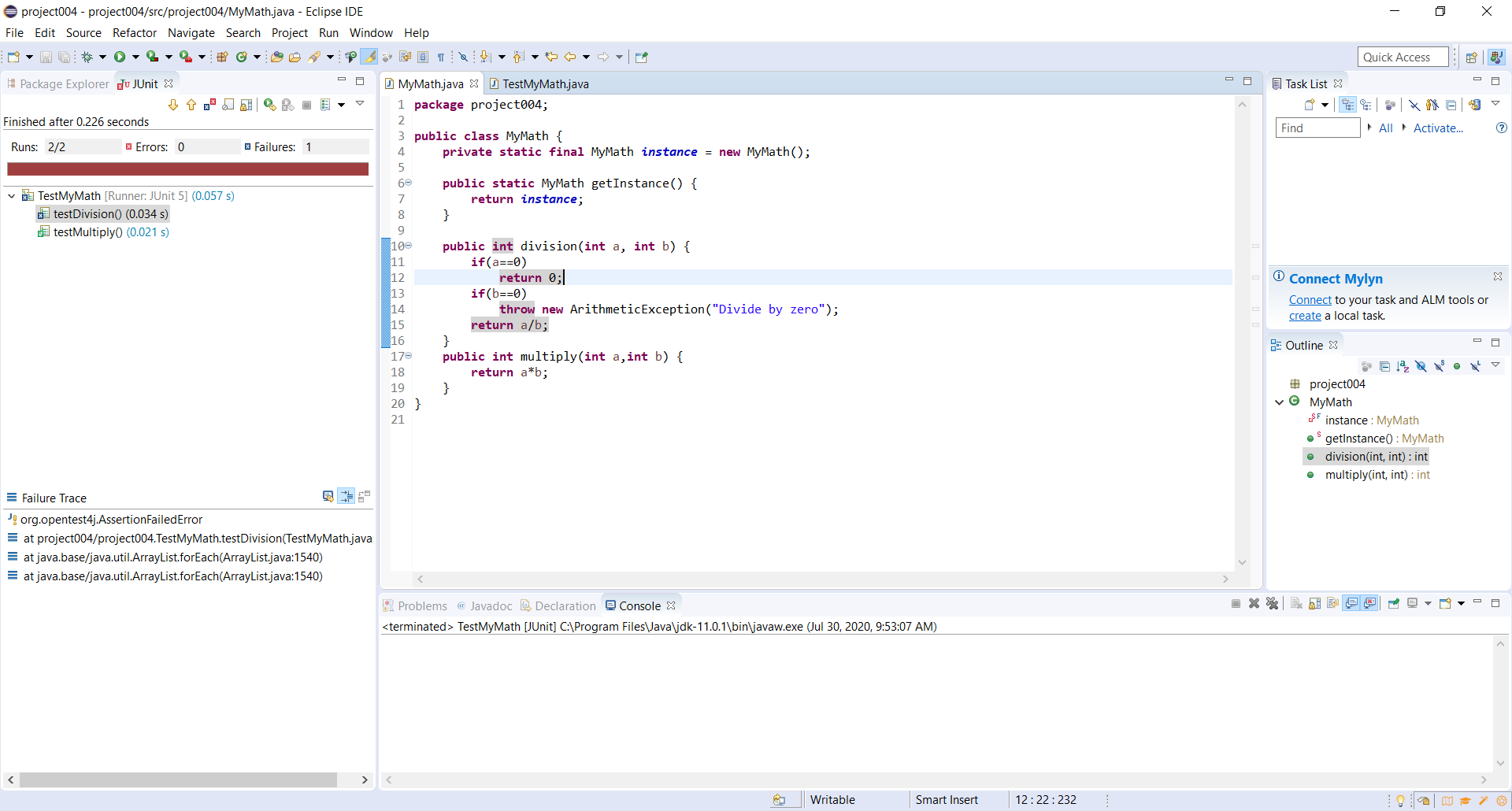


# **بررسی چند مثال :**

برای اضافه کردن library Junit به پروژه در Eclipse :

Java project -> Properties -> Java Build Path -> Libraries -> Add Library -> Junit 5

خالا در همین کد بالا شرطی را اضافه کردیم برای if(a==0) return 0; که خب غلطه چون حتی 0/0 را هم 0 برمیگرداند ! پس تست ما fail میشود . به این میگویند Collateral damage ! و به خاطر تست های خوبمون اشکال رو پیدا کردیم .



یک مثال دیگر :

**package** package5;

**import** **static** org.junit.Assert.\*;

**import** org.junit.\*;

**class** Business{

**public** **void** throwException() {

**throw** **new** RuntimeException("I throw an exception");

}

**public** **double** mutltiply(**double** a,**double** b) {

**return** a\*b;

}

}

**public** **class** TestFeatures {

**private** Business business;

@Before

**public** **void** setup() {

business = **new** Business();

}

@Test(expected=RuntimeException.**class**)

**public** **void** testException() {

business.throwException();

}

@Ignore

@Test

**public** **void** incompleteTest() {

*assertEquals*(1, 2);

}

@Test

**public** **void** testMultiply() {

**double** multiple = business.mutltiply(1.5, 2);

*assertEquals*(multiple, 3 ,0.0001);

}

}

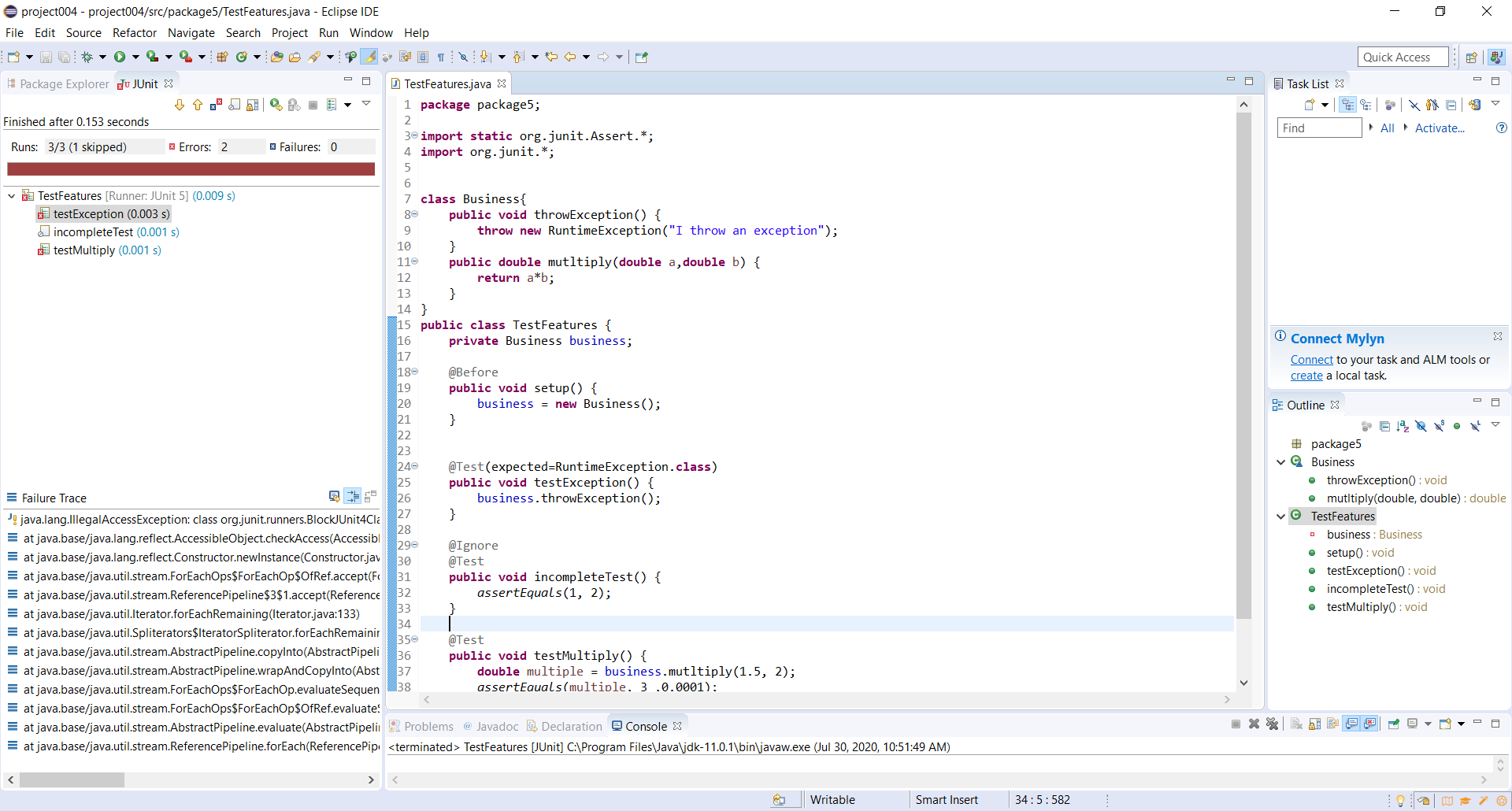
توجه : همانطور که بالا دیدیم ، میتوانیم برای @Test ، پارامتر هم در داخل ***پرانتز*** در نظر بگیریم . تا بگوییم که انتظار داریم در این تست ، این اتفاق بیفتد .

توجه : برای تست هایی که هنوز کامل نشده اند ، از @Ignore استفاده میکنیم .

توجه : داده های اعشاری به صورت تقریبی ذخیره میشوند لذا حاصل assertEquals برای ضرب 2 تا مقدار 1.5 و 2 نمیشود دقیقا 3 . پس برای اعداد حقیقی double باید یک بازه خطا هم پاس میدهیم .

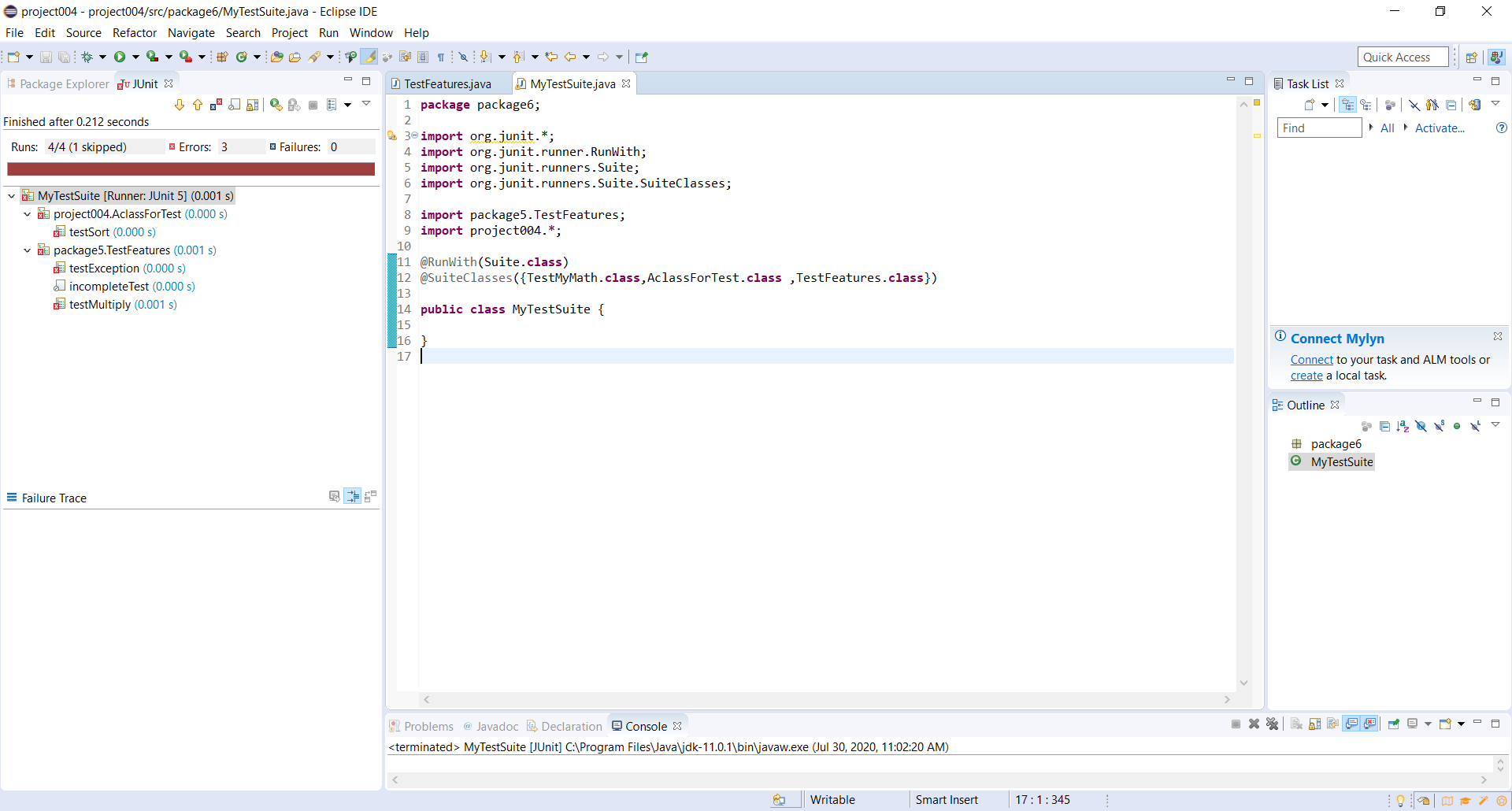
توجه : import static یعنی تمام متد های static آن کتابخانه import شوند

توجه : نمیدانم چرا و مشکلش چی بود که با اصلاحات هم مشکل حل نشد و fail ماند .



یک مثال دیگر از ساختن testSuite ها در Junit :

testSuite مجموعه ای از تست ها هست که قرار است اجرا شوند . با کمک این تست سوییت میتوانیم تست کیس ها را دسته بندی کنیم . مثلا :



برای این کار باید از @Runwith استفاده کنیم تا از امکانات suite برای این فایل استفاده شود .

و همچنین برای معرفی کلاس ها از @SuiteClasses استفاده میکنیم .

توجه : خود Junit جزوی از زبان جاوا نیست چون همانطور که دیدید ، از org ، import میکنیم نه از java .

# **جمع بندی :**

* ارتباط Maven و Junit مطالعه شود
* چارچوب Mock مطالعه شود – بخش هایی از unit را که وابستگی دارد به صورت بدلی در نظر میگیرد تا تست انجام شود .
* ابزار های محاسبه test coverage
* ابزار های گزارش دهی درباره وضعیت تست ها مثلا Jenkins
* عدم انجام آزمون واحد یک وام فنی است که باید بعدا آنرا پس دهیم به علاوه سود و هزینه اش ! (technical debt)
* آزمون واحد یک "باید" است
* کد بدون تست ، کد بدی هست
* میتوانیم یک نرم افزار مبتنی بر وب را درمقابل شبیه سازی حجم شدید درخواست های کاربر آزمایش کنیم.
* کیفیت نرم افزار
  + تضمین کیفیت : فرآیند تولید نرم افزار
  + کنترل کیفیت : یکیاز مراحل پایانی تولید که برخی جنبه ها را می سنجد
* معمولا در پروژه ها ، مجموعه تست ها در یک سرور اجرا میشوند :
  + به صورت زمانبندی شده و خودکار
  + بعد از هر commit
* test coverage :
  + این ابزار نشان میدهد که چه نسبتی از متن برنامه تحت تست قرار گرفته است
  + میزان exception هایی که مدنظر قرار گرفته است را محاسبه میکند
  + هر چه پوشش تست بیشتر باشد ، گزارش اشکالات کمتر خواهدبود و اگر برنامه ای اشکالات گزارش شده اش زیاد باشد باید بازنویسی شود
* ویژگی حرفه ای بودن تست :
  + تولید کد تست ، جدی گرفته شود
  + آزمون واحد تمام ویژگی های یک طراحی خوب را داشته باشد
    - encapsulation
    - coupling پایین
    - cohesion بالا
* Mock objects (اشیا بدلی)
  + بخش هایی که آن واحد مورد آزمون به آن وابسته است ، نباید در آزمون موثر باشد . در واقع هدف از آزمون واحد ، فقط تست کردن یک واحد است نه واحد های وابسته به آن .
  + راهکار برای حذف وابستگی ها استفاده از Mock است .
  + ابزار های این کار : Mockito و ....