

# **Esercitazioni Ing.Sw 2023-24**

Webex:

<https://politecnicomilano.webex.com/meet/gianenrico.conti>

10047232@polimi.it

Mese	Data	Attività	Ore	Argomento
Febbraio	20	-		Introduzione al corso
	20	esercitazione	3	Tools (IntelliJ, git, maven) + Testing (JUnit)
Marzo	5	esercitazione	3	Presentazione progetto + MVC
	12	lab	4	Verifica installazione tool + primo draft UML della parte di
	19	lab	4	Software design: UML + Implementazione classi modello
	26	esercitazione	3	Socket, Serializzazione, (distributed) MVC
Aprile	2	Pasqua		
	9	Lauree		
	16	lab	4	Implementazione classi complete modello + Testing
	23	lab	4	Software design: architecture
	30	lab	4	Comunicazione, Multi-threading, Protocol design
Maggio	7	esercitazione	3	Swing (accenni JavaFX)
	14	lab	4	Comunicazione, Multi-threading, Protocol design + client
	21	lab	4	GUIs: Swing
	28	lab	4	Discussione generale progetto e deployment

 In giallo consegna del documento di peer-review ai reviewer (si ricevono commenti la settimana successiva)

Lab 4 H

Ex 3 H

# **Es 1 tools**

## **IntelliJ, Maven, Git, JUnit**

(Credits to prof. Baresi, too)

IntelliJ is an Integrated Development Environment (IDE)

- – it contains a text editor
- – it is able to compile your code (through the Java SE JDK)
- – it allows you to debug and test your code
- IntelliJ is both open source (Community edition) and licensed (Ultimate edition)
- IntelliJ can be extended with plugins

## RECAP:

- Video di supporto (segue link)

- "Quasi" automatico

- Impostate JDK a "21"

- Evitare setup "sporchi"

(precedenti install.. mix sdk..etc.. \$PATH pasticciate...)

- Utilizzate solo:

  - "Solo" MAVEN (more about Maven / JavaFX later..)

- **Non** useremo IntelliJ alla "VStudio/Xcode/AndroidSDK"

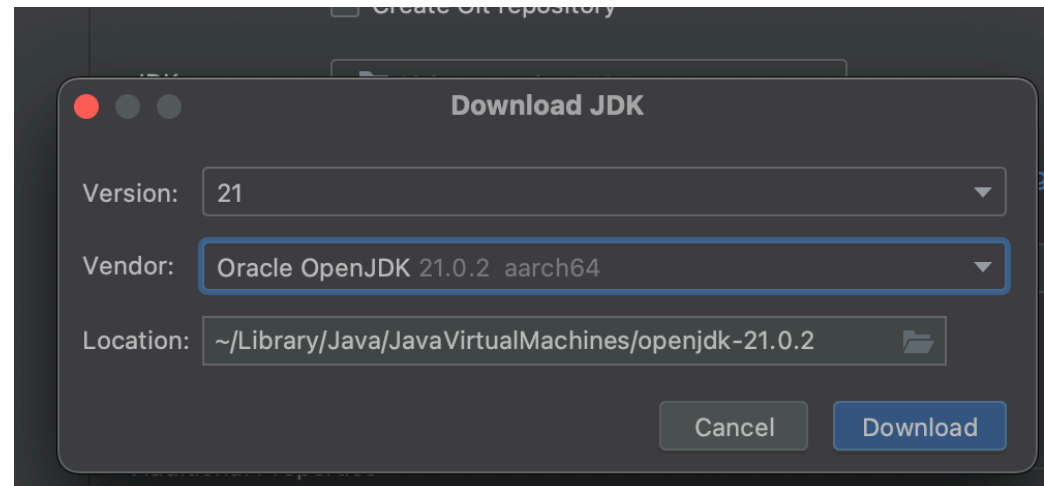
# IntelliJ setup-cleanup

6

Eliminare plugin non utilizzati (as per JetBrains help)

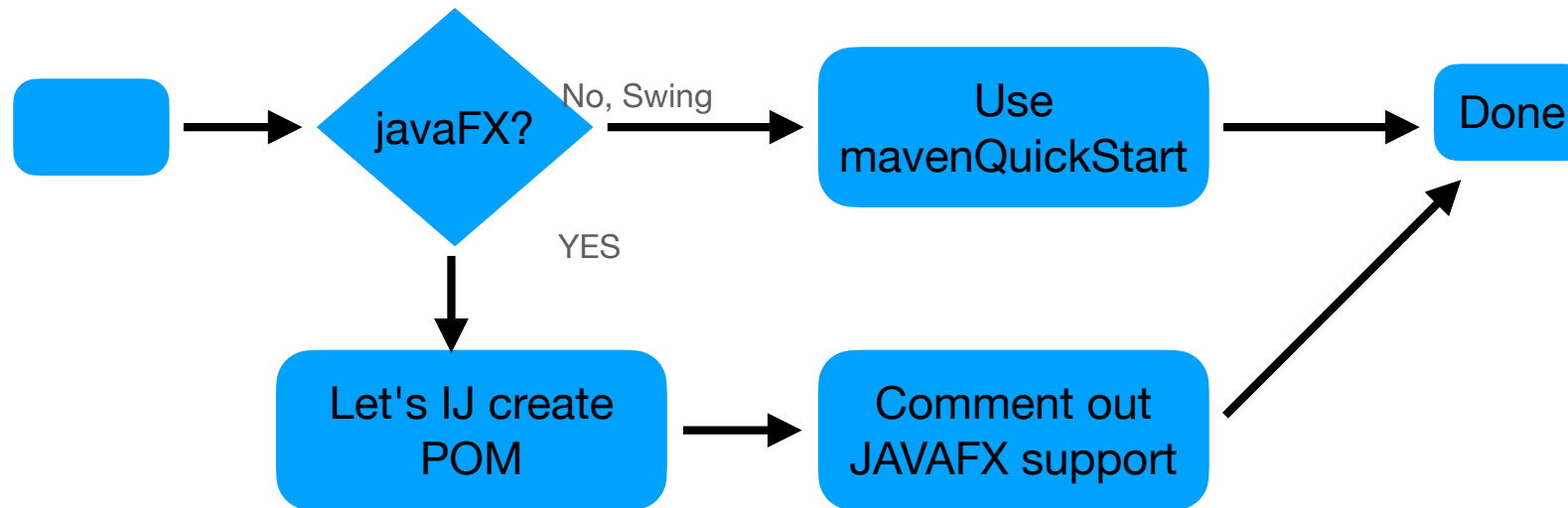
Useremo praticamente solo MAVEN (see "install" slides)

Check JDK:



Ma Eclipse? ...

(Slide a parte)



Logica:

- partire da maven -> javafx support spesso troppo complesso
- IntelliJ crea POM
- commentiamo codice di supporto x "dopo"

# **Maven**

Breve introduzione



# Automation

- Developers should only care of configurations
  - Tools should perform required actions
- Automation saves time and makes the process easily reproducible

# Good practices

- Separate code, tests, resources
- Platform independence
- Declarative dependency management
- Share resources via dependencies
- Predictable directory structure
- Test built artifacts

# Build tools

- Generate source code (if auto-generated code is used in the project)
- Generate documentation from the source code
- Compile source code
- Package compiled code into JAR files or ZIP files
- Install the packaged code on a server, in a repository or somewhere else

*MAVEN does it!*



<https://maven.apache.org>

Software projects are composed by several modules

- with several third-party libraries
- Manual lifecycle management is hard

Per gestirli si usa Maven

“Apache Maven is a project management tool”

Permette di gestire:

- Dependencies
- Building
- Testing
- Packaging
- Deployment
- Reports and Metrics
- Documentation

# How does Maven work?

- Configuration is done via a Project Object Model
- A POM file
  - Is an XML representation of the project
  - Manages dependencies
  - Configures plugins for building the software

- Fasi:
  - Validazione
  - Compilazione
  - Testing
  - Packaging
  - Installazione
  - Deploy

Ogni fase richiede la corretta esecuzione delle precedenti

Modalita' **standalone** (command line) o **Maven Tool Window** (GUI)

# Maven phases (1)

Validate:

- Maven checks that the project is properly configured
- Correctness of the configuration file
- Availability of everything needed by other phases

Compile:

- This phase only compiles all the java files in the project
- It is useful to separate dependencies needed to build the project with others needed at later phases

Test:

- This phase executes the tests, reporting the outcome
- If a test fails, the other phases blocks execution



# Maven phases (2)

## Package

- Creates a package (usually a jar) containing the compiled code and other resources

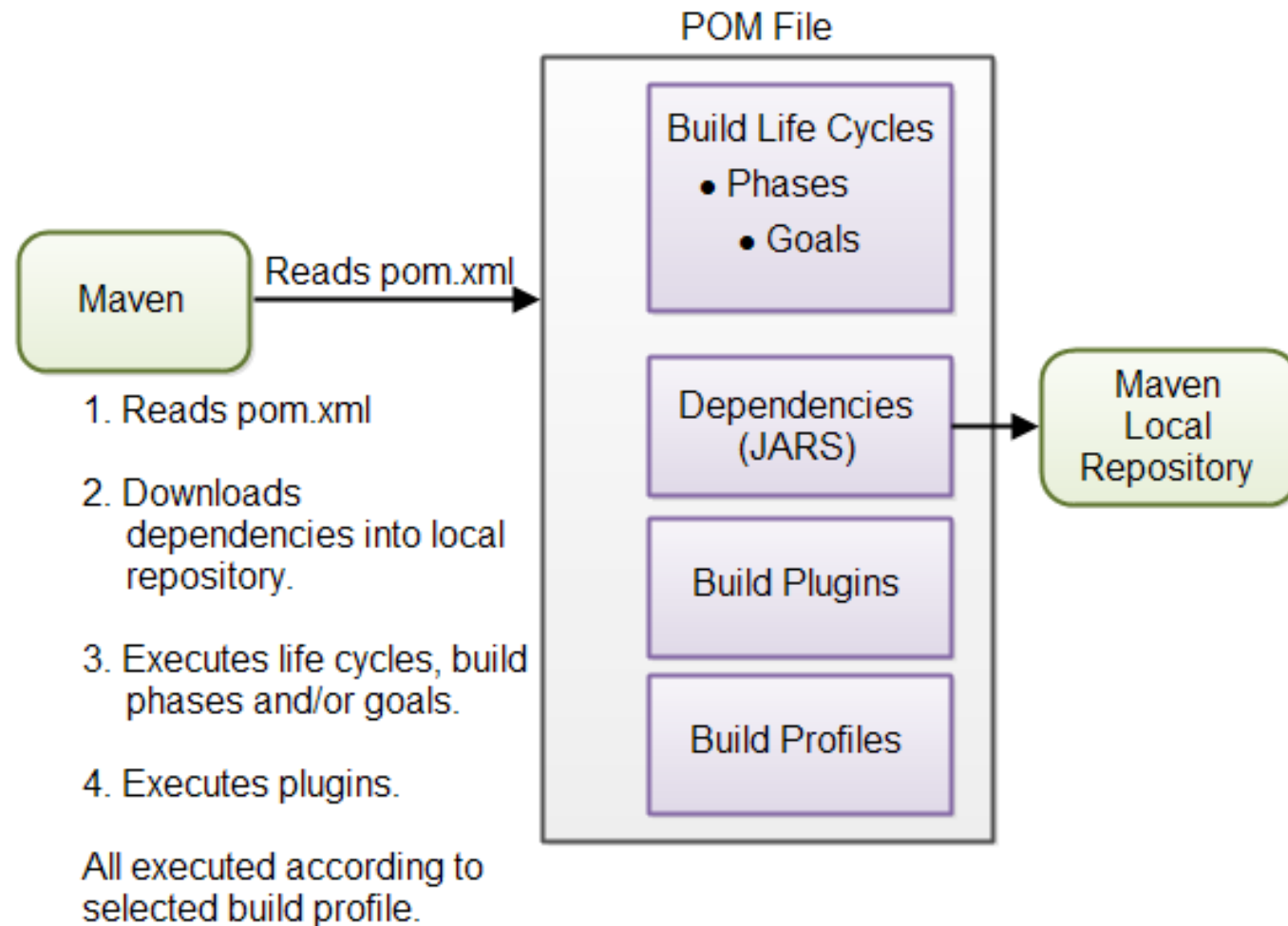
## Maven Install

- Copies the package to the local repository
- Once the project is installed in a repository it can be used as a dependency

## Deploy

- Publishes the package to a remote repository so that they can be made publicly available

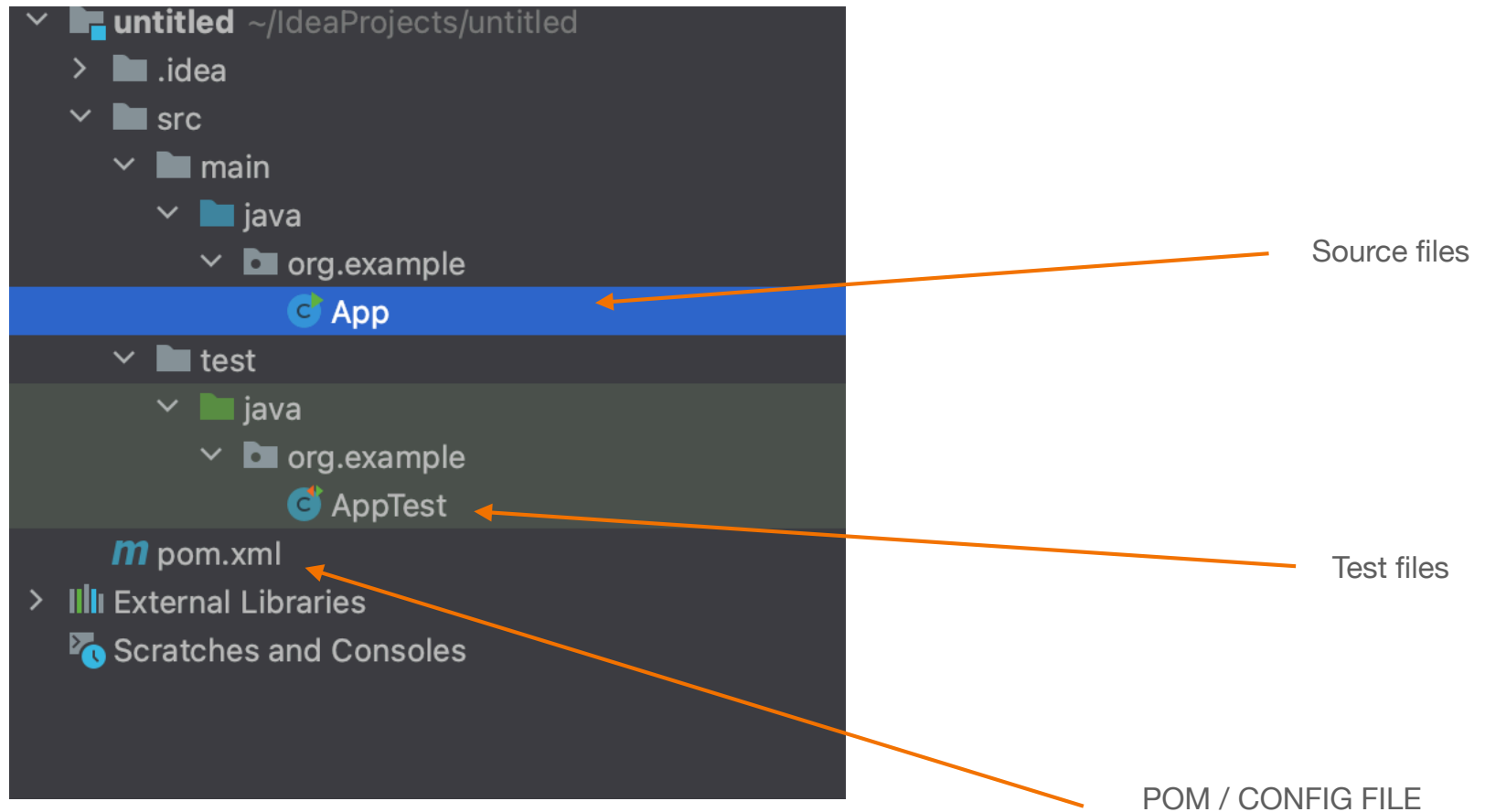
# In a nutshell



# Maven project structure

19

dettaglio:



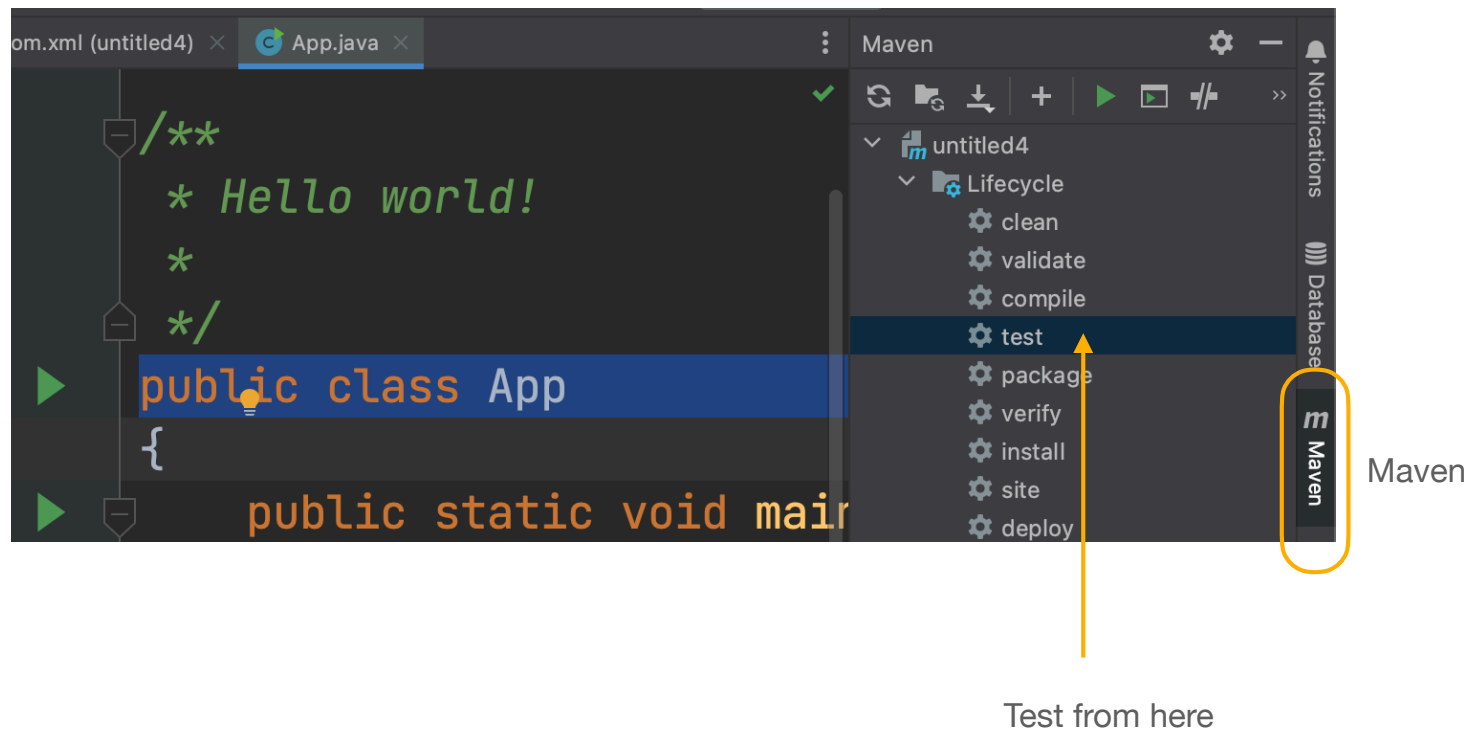
# Any given MAVEN proj...

Si era detto:

- Validazione
- Compilazione
- Testing
- Packaging
- Installazione
- Deploy

Maven TAB a DX...

# Any given MAVEN proj...



# Any given MAVEN proj...

Doppio click p.es. su Compile...

```
INFO] Scanning for projects...
```

```
....
```

```
[INFO] BUILD SUCCESS
```

```
[INFO] -----
```

```
[INFO] Total time: 1.038 s
```

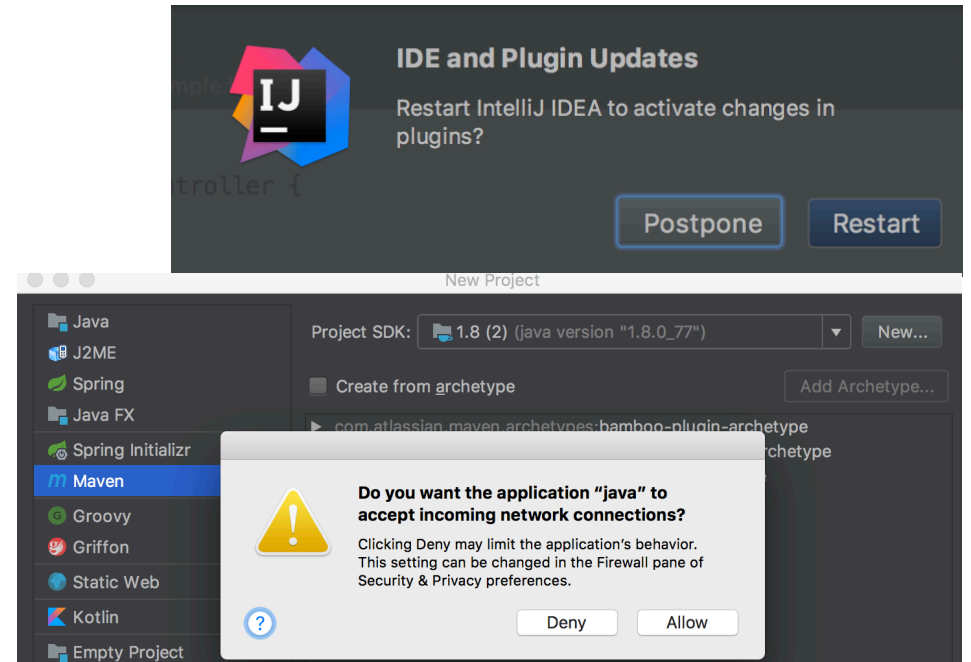
```
[INFO] Finished at: 2024-02-20T09:12:14+01:00
```

```
[INFO] -----
```

```
Process finished with exit code 0
```

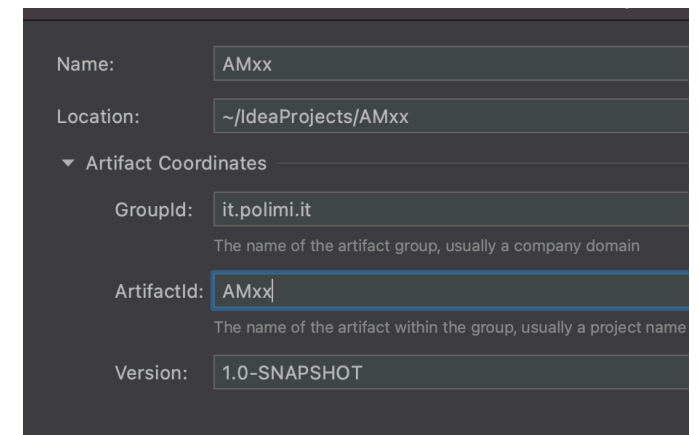
# MAVEN tips and tricks...

- IntelliJ ha plugin Maven
- Abilitando plug in puo' apparire:
- More during 1st lab..



**ID GRUPPO:**      **it.polimi.ingsw**

**Artifact ID:**      **AM<vs. gruppo>**



# Any given MAVEN proj...

Ci aspettiamo un progetto Maven:

- Compilabile
- Testabile
- Eseguibile
- Senza errori





# Git

Breve introduzione

# The Birth of Git

- Git is a *distributed* version control system
  - Differently from *centralized* version control systems such as SVN
- Created by Linus Torvalds in 2005 for the development of the Linux kernel
- Git is free software

# Git in a nutshell

Git è un Version Control System (VCS) decentralizzato  
Cos'è un sistema di controllo versione?

*Un software che serve a tracciare lo sviluppo di un progetto*



Ciò offre numerose comodità in fase di sviluppo



# Git in a nutshell

Perché un sistema di controllo versione?

Non è sufficiente la funzione UNDO? / ZIP...?

Risposta breve: **No**

# Git basics

- Idea di fondo: salvare lo stato di ogni file del progetto (come se si facesse una foto del progetto in un certo istante.)
- Tali istantanee sono detti **commit**
- Key Idea: Albero dei commit, dove lo storico di sviluppo può separarsi in più branch

questi rami possono poi tornare a fondersi insieme con una operazione di merge

# Installazione

- il sito di riferimento è <https://git-scm.com>
  - Windows: <https://desktop.github.com>
  - Linux: apt install git
  - MacOS: (nativo)
- Interfaccia base da linea di comando
- Spesso integrato negli IDE

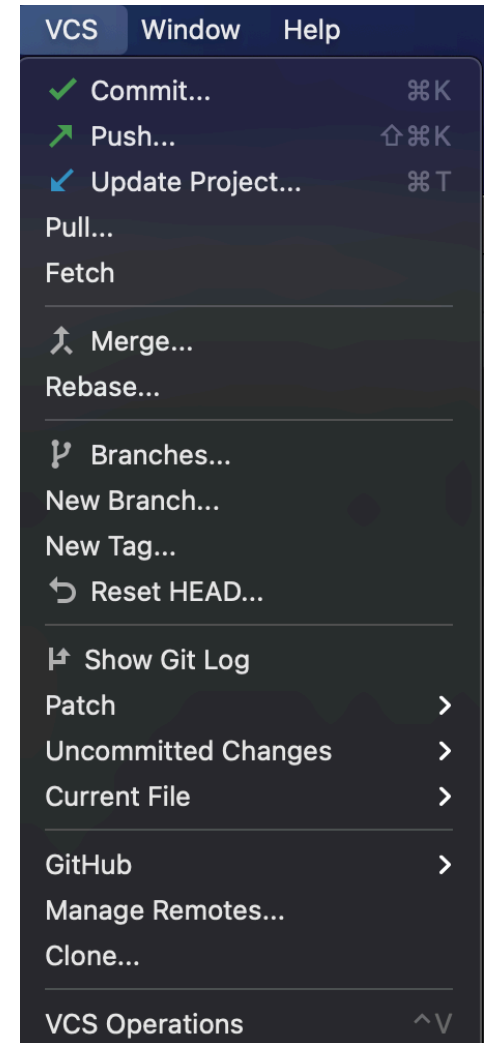


POLITECNICO  
DI MILANO

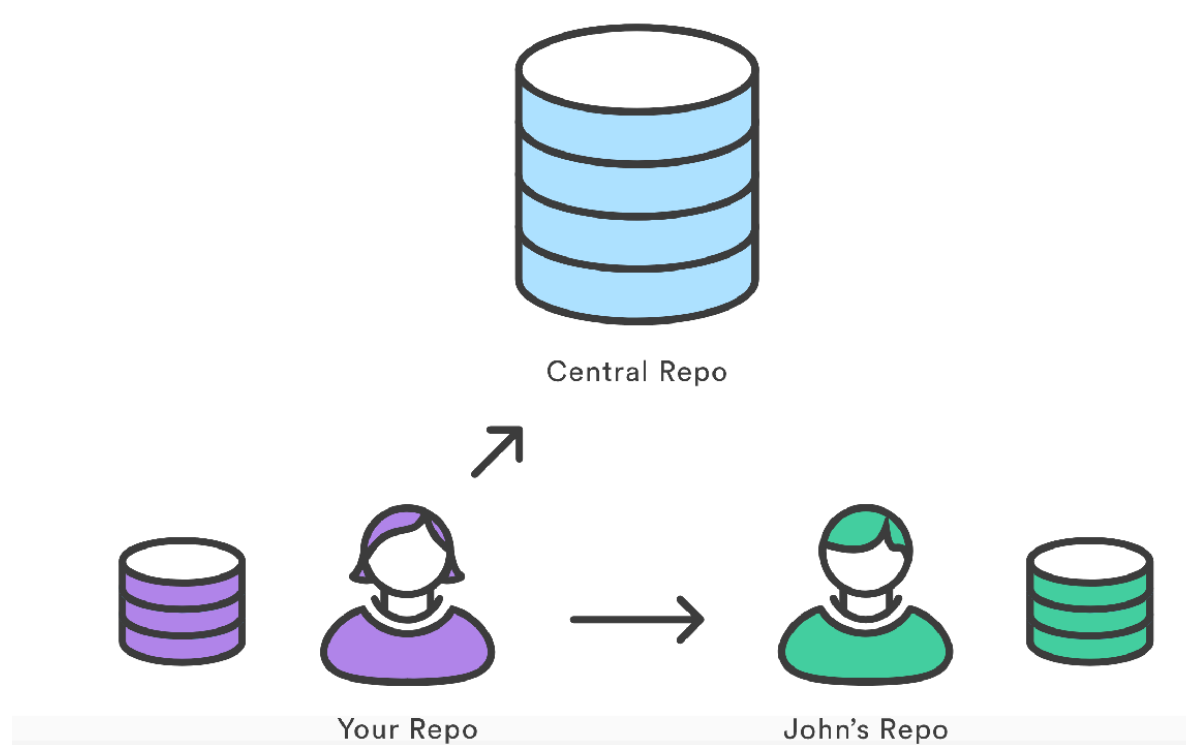


# Installazione

- Spesso integrato negli IDE
- IntelliJ:



# Workflow tipico Git



- Gli utenti di un repository possono inviare modifiche **simultaneamente**
  - A patto che le modifiche non siano sulle stesse linee di codice
- Possono condividere tra loro le modifiche fatte indipendentemente



# Stato dei files

Per ogni file 4 stati:

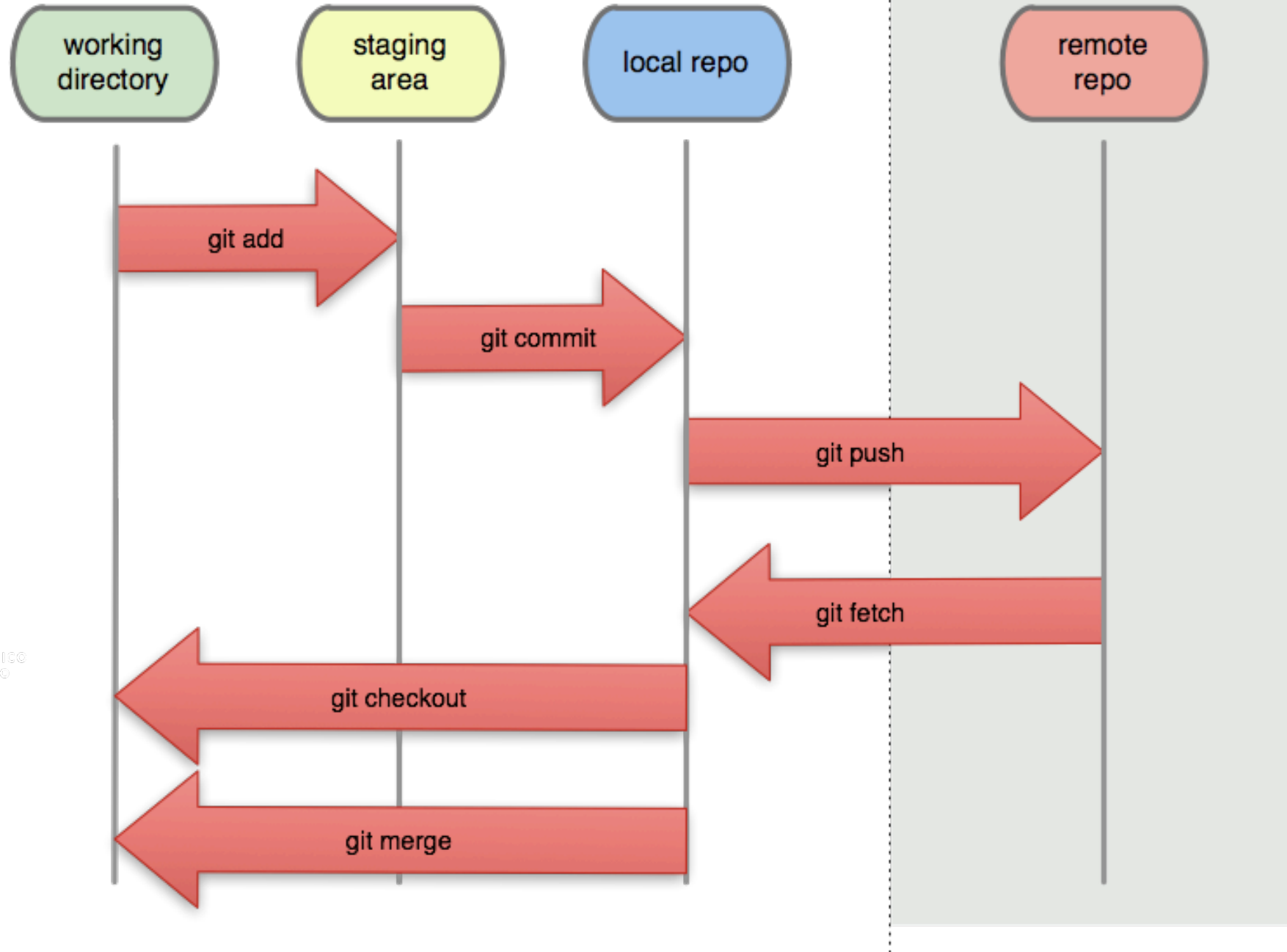
- **Untracked** - non tracciato, non è un vero e proprio stato
- **Modified** - il file è stato modificato
- **Staged** - il file è stato contrassegnato per essere salvato nel database
- **Committed** - una copia del file è salvata al sicuro nel database Git

*I commit possono essere trasferiti in repo remoto (i.e. push)*



## Local

## Remote



POLITECNICO  
DI MILANO



# Comandi

- clonazione repo Github: *git clone*
- aggiunta modifiche: *add, rm, mv*
- Salvataggio modifiche e sincronizzazione: *commit, push, pull*
- conflitto: *merge*



POLITECNICO  
DI MILANO



# Lottare con Git

- Git è un tool nato, fatto e usato da professionisti
- Quindi, se vi dà errore ... HA RAGIONE LUI!!!
  - il 99.9% delle volte
- L'opzione *--force* di molti comandi è IL modo per fare ulteriori danni
- Cercare nella documentazione (man, git-scm.com) e su StackOverflow



POLITECNICO  
DI MILANO



# Howto: riparare commit distruttivo

- `git add *.java`
- `git commit -m "refactoring innocente; state sereni :)"`
- `git push`
- QUALCOSA NON TORNA...
- I COMPAGNI TELEFONANO ARRABBIATI
- OK, E' COLPA MIA: "Alberto broke the build"
- MAIL DI SCUUSE: "scusate :( fixo e pusho"
- cosa faccio?
- `git revert HEAD`
- `git push`
- MAIL "pullate pure, ho fatto revert grazie a Git :)"

la prima tentazione è cancellare il commit remoto che ho appena fatto, e che tutti hanno pullato

la history avanza SEMPRE: git revert crea un NUOVO commit che annulla gli effetti del (dei) precedente(i)



POLITECNICO  
DI MILANO



# GitHub

- Ogni studente deve creare un account personale GitHub student
  - Con e-mail istituzionale
- Una volta stabilito il gruppo di appartenenza, uno dei membri del gruppo crea un repository privato denominato ing-sw-2021-cognome1-cognome2-cognome3  
***PLS NO redwolf2001 or similar..***
- Invitate al repo tutti i membri del gruppo e i responsabili:
  - @ingconti
  - barbara.schinaia@gmail.com
  - (seguono Tutor...)

# **Test e jUnit**

Breve introduzione

# Test

- Si fanno esperimenti con il programma allo scopo di scoprire eventuali errori
  - Si campionano i comportamenti
  - Fornisce indicazioni parziali relative al particolare esperimento
  - Il programma è effettivamente provato solo per quei dati
- Il test è una tecnica dinamica rispetto alle verifiche statiche fatte dal compilatore



# Test black-box e white-box testing

- Black-box o *funzionale*
  - Casi di test determinati in base a ciò che il componente deve fare
    - La sua *specifica*
- White-box o *strutturale*
  - Casi di test determinati in base a come il componente è implementato
    - Il suo *codice*

# **Dijkstra (1972)**

Program testing can be used to show the presence of bugs, but never to show their absence



“Whenever you are tempted to type something into a **print** statement or a debugger expression, write it as a test instead.”

Martin Fowler

# JUnit (1)

- JUnit è un framework per scrivere test
  - Proposto da Erich Gamma (Design Patterns) e Kent Beck (Extreme Programming)
- JUnit usa la riflessività di Java per
  - Integrare codice e test
  - Eseguire test e test suite

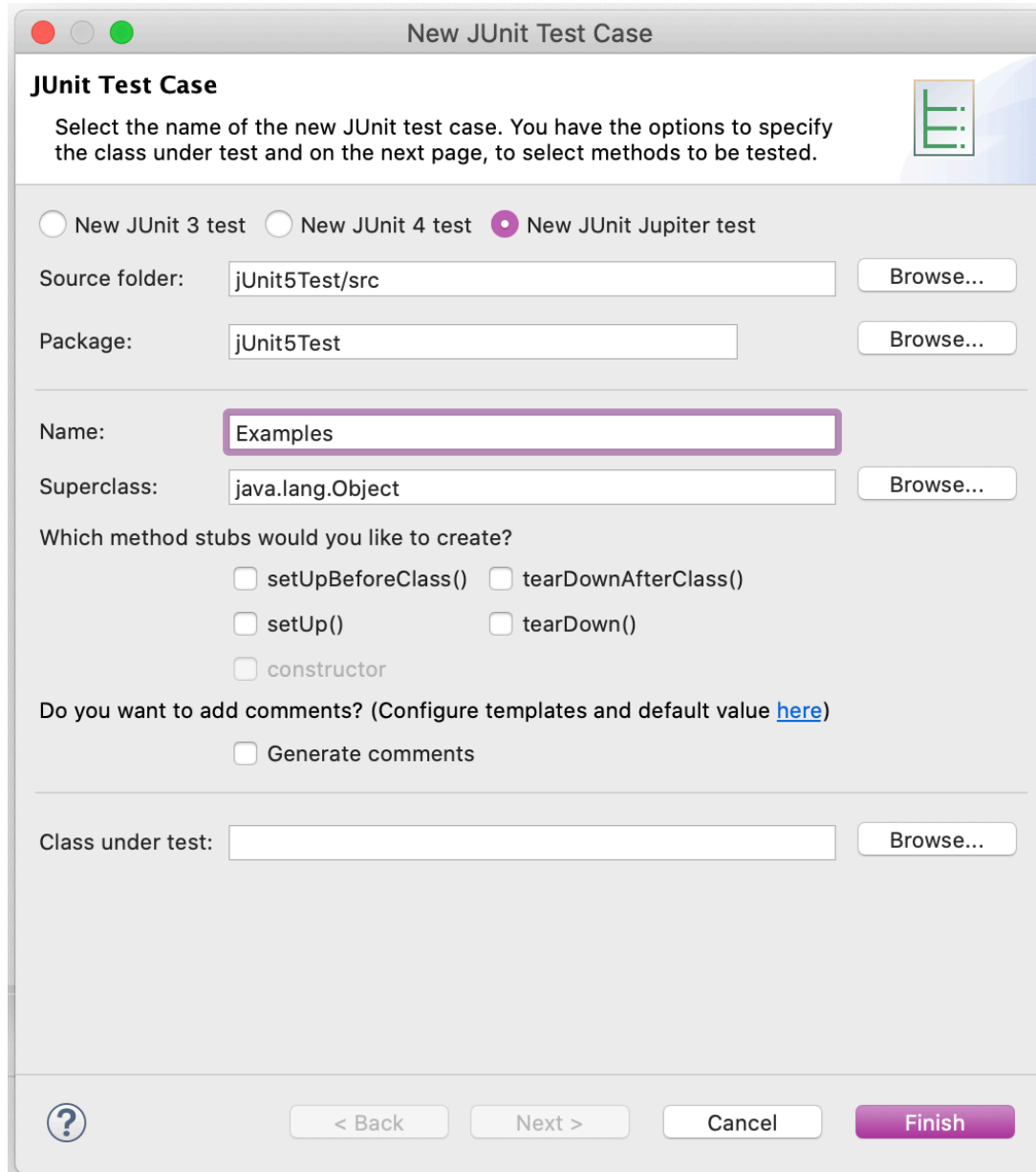
# JUnit (2)

- JUnit non è parte di nessun SDK, ma tutti i principali IDE lo includono
- JUnit 5 = Platform + Jupiter + Vintage
- Estendibile con framework esterni
  - Ad esempio, AssertJ

# Idea di base

- Data una classe Foo, creare un'altra classe FooTest per eseguire il test della precedente attraverso opportuni metodi
  - Ogni metodo è un test
- JUnit mette a disposizione metodi assert per la scrittura dei test
  - Questi metodi vanno chiamati nei metodi test per controllare quanto di interesse (oracolo)

# JUnit e Eclipse



The screenshot shows the 'New JUnit Test Case' dialog box in Eclipse. The title bar reads 'New JUnit Test Case'. The main heading is 'JUnit Test Case'. Below it, a text box says: 'Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.' To the right of this text is a small icon of a document with a green 'E' on it.

There are three radio buttons for selecting the type of test case:

- ☐ New JUnit 3 test
- ☐ New JUnit 4 test
- ☒ New JUnit Jupiter test

Below these are two text fields with 'Browse...' buttons:

Source folder:

Package:

---

Name:  (This field is highlighted with a purple border)

Superclass:

Which method stubs would you like to create?

- ☐ setUpBeforeClass() ☐ tearDownAfterClass()
- ☐ setUp() ☐ tearDown()
- ☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

- ☐ Generate comments

---

Class under test:

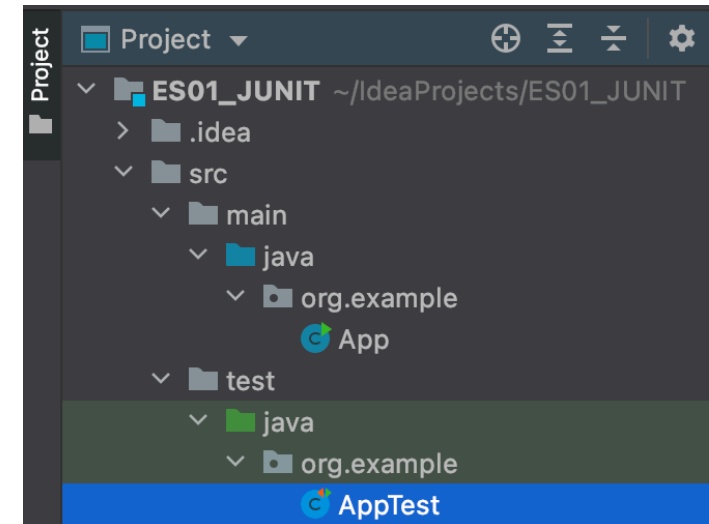
At the bottom, there is a row of buttons: a question mark icon, '< Back', 'Next >', 'Cancel', and a purple 'Finish' button.



# JUnit e IntelliJ

(<https://junit.org/junit5/>)

Solito progetto ...



Cartella x tests é usualmente gia settata da Maven quickstart

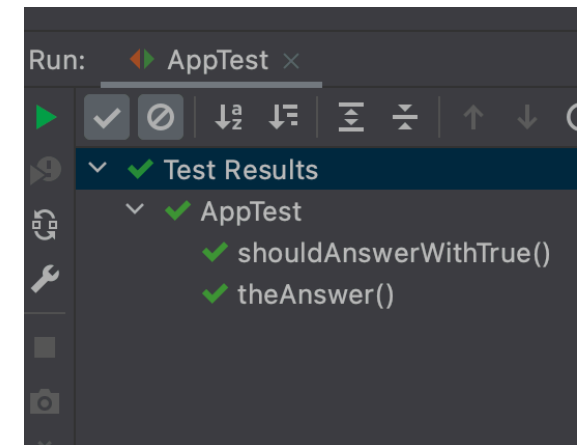


# Un primo test

```
package ....
```

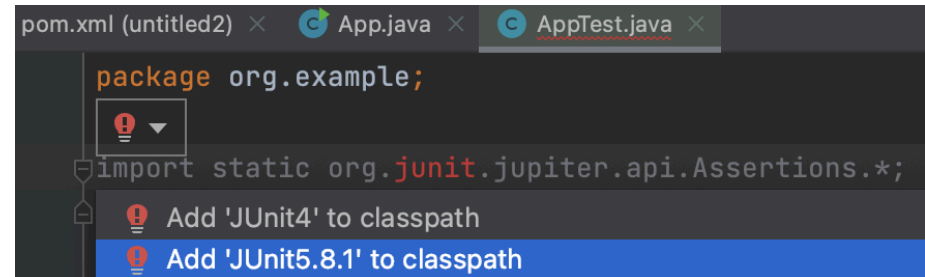
```
import static org.junit.jupiter.api.Assertions.*;  
import org.junit.jupiter.api.Test;
```

```
class AppTest {  
    @Test  
    void theAnswer() {  
        assertEquals(42, 22 + 20);  
    }  
}
```



# Un primo test :(

- Usual Alt enter..



And fix also version... (if needed)

```
Module untitled1 SDK 19 does not support source version 1.5.  
Possible solutions:  
- Downgrade Project SDK in settings to 1.5 or compatible. Open project settings.  
- Upgrade language version in Maven build file to 19. Update pom.xml and reload the project.
```

# Un primo test :(

- Oppure a MANO:

```
<properties>
```

```
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

```
<maven.compiler.source>1.8</maven.compiler.source>
```

```
<maven.compiler.target>1.8</maven.compiler.target>
```

```
</properties>
```

...

```
<!-- junit 5 -->
```

```
<dependency>
```

```
<groupId>org.junit.jupiter</groupId>
```

```
<artifactId>junit-jupiter</artifactId>
```

```
<version>RELEASE</version>
```

```
<scope>test</scope>
```

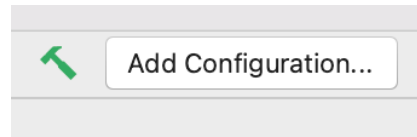
```
</dependency>
```



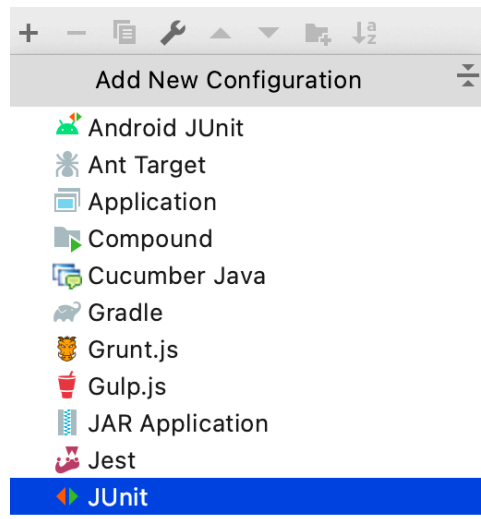
# Lavoriamo "comodi": Configuration

## IJ Test

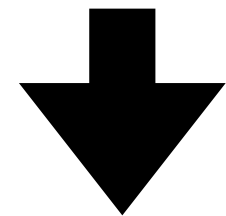
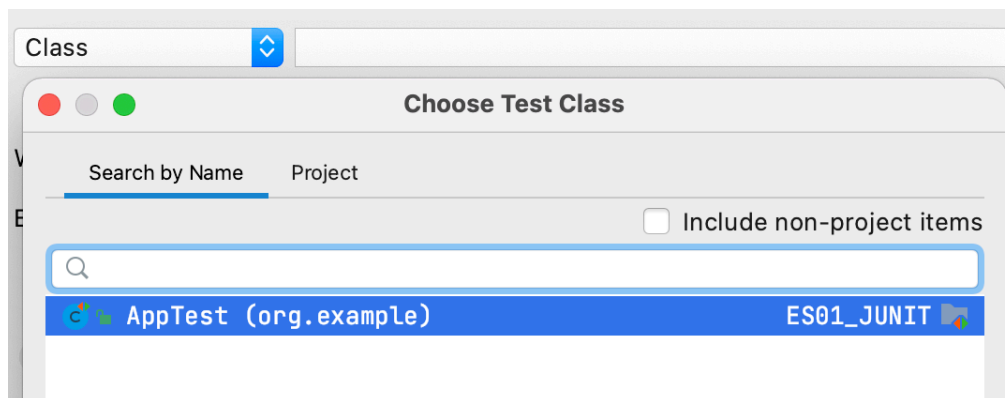
1) Add



2) Junit



3) Set class



# @Test

- **@Test** indica un metodo test
- Questi metodi
  - Non possono essere **private** o **static**
  - Possono dichiarare parametri
- Jupiter consente le meta-annotazioni
- Assertion e Assumption
- Junit5 non supporta le test suite come Junit4

# Alcune asserzioni

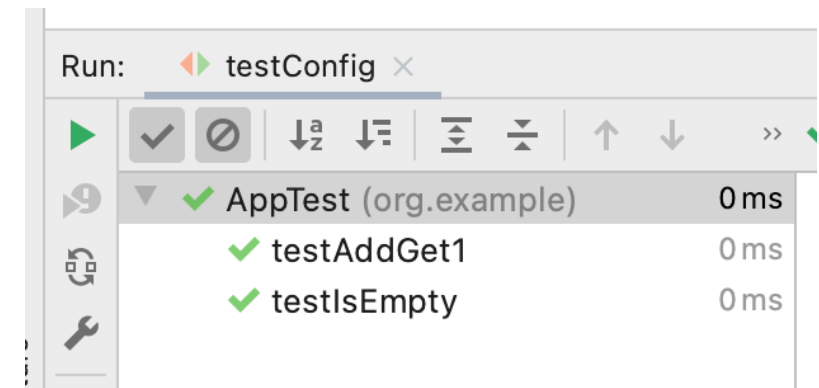
<code>assertTrue(<b>test</b>)</code>	Fallisce se il test booleano è false
<code>assertFalse(<b>test</b>)</code>	Fallisce se il test booleano è true
<code>assertEquals(<b>expected</b>, <b>actual</b>)</code>	Fallisce se i due valori non sono uguali
<code>assertSame(<b>expected</b>, <b>actual</b>)</code>	Fallisce se i due oggetti non sono identici (==)
<code>assertNotSame(<b>expected</b>, <b>actual</b>)</code>	Fallisce se i due oggetti sono identici (==)
<code>assertNull(<b>value</b>)</code>	Fallisce se il valore non è null
<code>assertNotNull(<b>value</b>)</code>	Fallisce se il valore è null
<code>fail()</code>	Il test fallisce immediatamente

- Ogni metodo accetta anche un parametro messaggio:
  - Esempio: `assertEquals(atteso, effettivo, messaggio)`



# Proviamo ArrayList

```
class TestArrayIntList {  
    @Test  
    public void testAddGet1() {  
        ArrayList<Integer> list = new ArrayList<Integer>();  
        list.add(42); list.add(-3); list.add(15);  
  
        assertEquals(Integer.valueOf(42), list.get(0));  
        assertEquals(Integer.valueOf(-3), list.get(1));  
        assertEquals(Integer.valueOf(15), list.get(2));  
    }  
  
    @Test  
    public void testIsEmpty() {  
        ArrayList<Integer> list = new ArrayList<Integer>();  
        assertTrue(list.isEmpty());  
        list.add(123);  
        assertFalse(list.isEmpty());  
        list.remove(0);  
    }  
}
```



# Meta-annotazioni

- Stanno in `org.junit.jupiter.api`
- Le più comuni sono
  - `@BeforeEach`
  - `@AfterEach`
  - `@BeforeAll` (il metodo deve essere static)
  - `@AfterAll` (il metodo deve essere static)
  - `@DisplayName`
  - `@Tag`
  - `@Disabled`
- Altre più strane sono
  - `@Nested`
  - `@ParameterizedTest`
  - `@RepeatedTest`



# @DisplayName

```
import java.util.concurrent.ThreadLocalRandom;
import java.time.Duration;

class FirstJupiterTest {
    @Test
    @DisplayName("First test")
    void allYourBase() {
        assertAll(
            () -> assertTrue(true),
            () -> assertEquals("42", "4" + "2"),
            () -> assertTimeout(Duration.ofSeconds(1), () -> 42 * 42)
        );
    }

    @Test
    @DisplayName("Second test")
    void toInfinity() {
        Integer result = assertDoesNotThrow(() ->
            ThreadLocalRandom.current().nextInt(42)); assertTrue(result < 42);
    }
}
```

# @TestInstance

**@TestInstance** is a type-level annotation that is used to configure the **lifecycle** of test instances for the annotated test class or test interface.



**By default, both JUnit4/5 create a new instance of the test class before running each test method.**

# @TestInstance

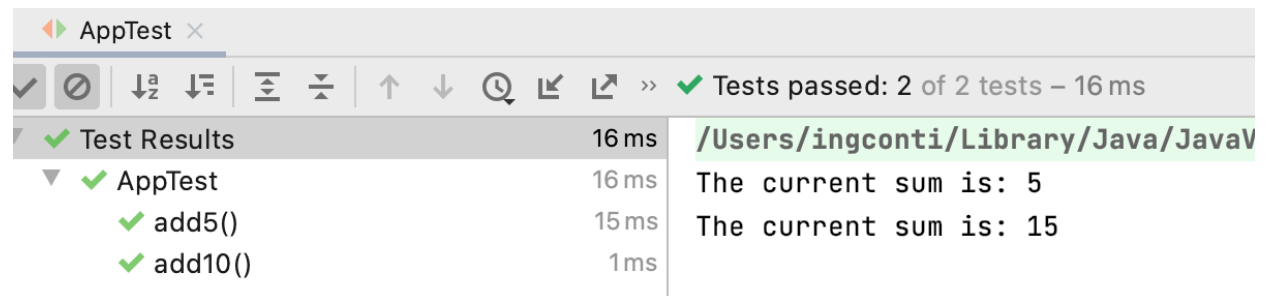
```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInstance;

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class AppTest {
    private int sum = 0;

    @Test
    void add5() { sum += 5; }

    @Test
    void add10() { sum += 10; }

    @AfterEach
    void tearDown() {
        System.out.println("The current sum is: " + sum);
    }
}
```



AppTest			Tests passed: 2 of 2 tests – 16 ms
Test Results			
AppTest	16 ms	/Users/ingconti/Library/Java/JavaV	
add5()	15 ms	The current sum is: 5	
add10()	1 ms	The current sum is: 15	

*Provate a rimuovere/commentare la riga*

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
```

..

# @TestInstance

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
...
```

Le istanze durano x tutta la vita della Classe:

La v. NON è una nuova v, conserva il suo valore fino a fine test: sum 15

Se commentiamo:

```
...
```

The current sum is: 5

The current sum is: 10

Process finished with exit code 0

# @Nested

@Nested is used to signal that the annotated class is a nested, non-static test class (i.e., an *inner class*) that can share setup and state with an instance of its **enclosing class**.



```
@DisplayName("An ArrayList")
class NestedTest {
    private ArrayList<String> list;
```

```
@Nested
@DisplayName("when new")
class WhenNew {
    @BeforeEach
    void createNewList() {list = new ArrayList<>();}
    @Test
    @DisplayName("is empty")
    void isEmpty() {assertTrue(list.isEmpty());}
```

```
@Nested
@DisplayName("after adding an element")
class AfterAdding {
    String anElement = "an element";

    @BeforeEach
    void addAnElement() {list.add(anElement);}
    @Test
    @DisplayName("it is no longer empty")
    void isEmpty() {assertFalse(list.isEmpty());}
}
```

# @Nested

Finished after 0.173 seconds

Runs: 2/2    ✖ Errors: 0    ☒ Failures: 0

```
▼ [✓] An ArrayList [Runner: JUnit 5] (0.024 s)
  ▼ [✓] when new (0.024 s)
    [✓] is empty (0.013 s)
    ▼ [✓] after adding an element (0.011 s)
      [✓] it is no longer empty (0.011 s)
```

# Test parametrici

- Altre sorgenti
  - Stringhe CSV
  - file CSV
  - interfaccia ArgumentsProvider

```
@ParameterizedTest
@ValueSource(ints = {2, 4, 6, 8, 9})
void evens(int value) {
    assertEquals(0, value % 2);
}
```

```
@ParameterizedTest
@MethodSource("supplyOdds")
void odds(int value) {
    assertEquals(1, value % 2);
}
```

```
private static Stream<Integer>
supplyOdds() {
    return Stream.of(1, 3, 5, 7, 9, 11);
}
```

Finished after 0.24 seconds

Runs: 11/11    Errors: 0    Failures: 1

ParametersTest [Runner: JUnit 5] (0.069 s)

odds(int) (0.041 s)

[1] 1 (0.041 s)

[2] 3 (0.003 s)

[3] 5 (0.004 s)

[4] 7 (0.002 s)

[5] 9 (0.006 s)

[6] 11 (0.009 s)

evens(int) (0.002 s)

[1] 2 (0.001 s)

[2] 4 (0.002 s)

[3] 6 (0.003 s)

[4] 8 (0.003 s)

[5] 9 (0.009 s)

Failure Trace

org.opentest4j.AssertionFailedError: expected: <0> but was: <1>

at nested.ParametersTest.evens(ParametersTest.java:15)

# Test ripetuti

```
@RepeatedTest(value = ....)
```

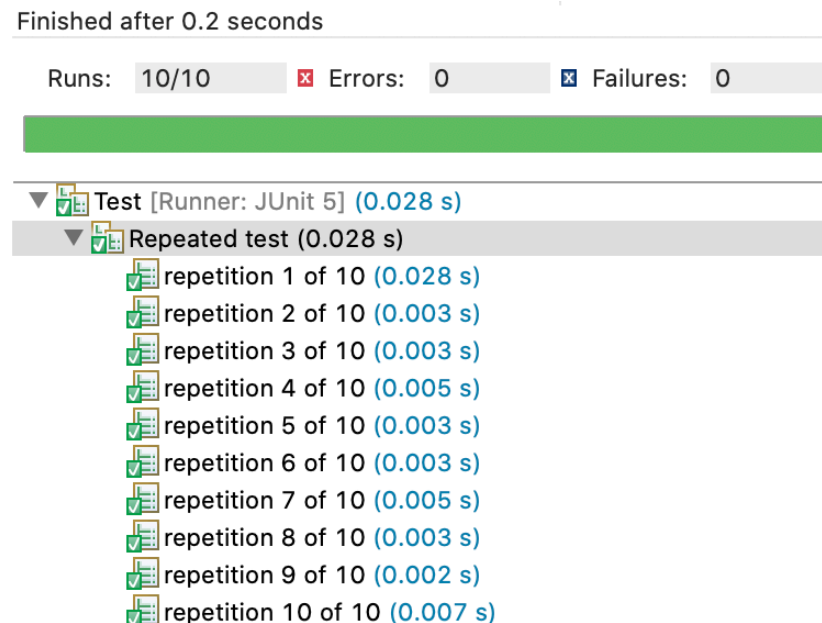
Each invocation of the repeated test behaves like the execution of a regular **@Test** method with full support for the same lifecycle callbacks and extensions. In addition, the current repetition and total number of repetitions can be accessed by having the **RepetitionInfo** injected.





# Test ripetuti

```
@DisplayName("Repeated test")
@RepeatedTest(value = 10)
void repeatedTestWithInfo(RepetitionInfo repetitionInfo) {
    assertTrue(repetitionInfo.getCurrentRepetition() <= repetitionInfo.getTotalRepetitions());
    assertEquals(10, repetitionInfo.getTotalRepetitions());
}
```



# Enabled and Disabled

```
import org.junit.jupiter.api.Disabled;  
import org.junit.jupiter.api.condition.DisabledOnJre;  
import org.junit.jupiter.api.condition.EnabledOnOs;  
import org.junit.jupiter.api.condition.OS;  
import org.junit.jupiter.api.condition.JRE;
```

```
@Test
```

```
@Disabled("for demonstration purposes")
```

```
@EnabledOnOs({OS.MAC, OS.LINUX})
```

```
@DisabledOnJre(JRE.JAVA_12)
```

# Eccezioni

```
class ExceptionTest {  
    @Test  
    void assertThrowsException() {  
        String str = null;  
        assertThrows(IllegalArgumentException.class, () -> {Integer.valueOf(str);});  
    }  
  
    @Test  
    void exceptionTesting() {  
        Exception exception;  
  
        exception = assertThrows(ArithmeticException.class, () -> Calculator.divide(1, 0));  
        assertEquals("/ by zero", exception.getMessage());  
    }  
}
```

# Timeout

```
@BeforeEach @Timeout(5)
void setUp() {
    // fails if execution time exceeds 5 seconds
}
```

```
@Test
@Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
void failIfExecutionTimeExceeds100Milliseconds() {
    // fails if execution time exceeds 100 milliseconds
}
```

```
@Test
void timeoutNotExceeded() {
    assertTimeout(Duration.ofMinutes(2), () -> {
        // Perform task that takes less than 2 minutes.
    });
}
```

# Assunzioni

- Se l'ipotesi non è valida, l'esecuzione del test viene abortita

```
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.Assumptions.*;
import org.junit.jupiter.api.Test;
```

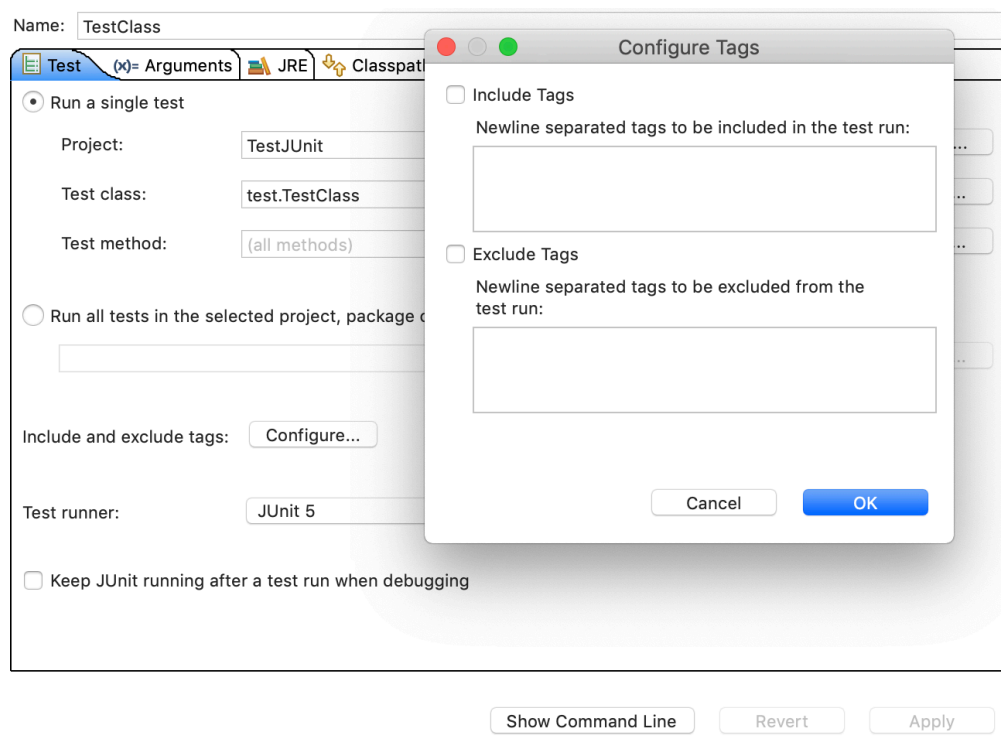
```
class AssumptionTest {
    @Test
    void trueAssumption() {
        assumeTrue(5 > 1);
        assertEquals(5 + 2, 7);
    }

    @Test
    void falseAssumption() {
        assumeFalse(5 < 1);
        assertEquals(5 + 2, 7);
    }
}
```

```
    @Test
    void assumptionThat() {
        String someString = "Just a string";
        assumingThat(someString.equals("Just a string"),
            () -> assertEquals(2 + 2, 4));
    }
}
```

# Tag

- Su classe e su metodo
- includeTags e excludeTags



```
class TestClass {  
    @Test  
    @Tag("development")  
    @Tag("production")  
    void testA() {}  
  
    @Test  
    @Tag("development")  
    void testB() {}  
  
    @Test  
    @Tag("development")  
    void testC() {}  
}
```

# Un esempio Black Box

- Vi hanno dato una semplice classe Calculator
- Dovete testare il fattoriale
- (E magari poi aprire.. diventa **white**...)
- Casi noti:
- Input 0 e neg...
- Input 10
- Basta? (proviamo..)

# Un esempio Black Box II

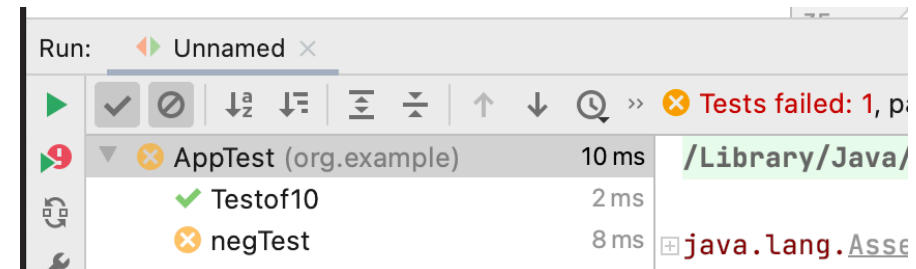
```
public void negTest()  
{  
    Calculator c = new Calculator();  
    assertTrue( c.Factorial(-1) > 0 );  
}  
@Test  
public void TestOf10()  
{  
    Calculator c = new Calculator();  
    assertTrue( c.Factorial(10) == 3628800 );  
}
```



# Un esempio Black Box III

Ok, un fallimento.

Fixiamo codice: (WHITE...)



```
int Factorial( int number ) {  
    return number <= 1 ? number : Factorial( number - 1 ) * number;  
}
```

Diventa:

```
int Factorial( int number ) {  
    return number <= 1 ? 1 : Factorial( number - 1 ) * number;  
}
```

# Un esempio Black Box IV

Run.. ok.

Ma siamo sicuri?

Aggiungiamo:

```
@Test
    public void TestOf18()
    {
        Calculator c = new Calculator();
        assertTrue( c.Factorial(18) > 0 );
    }
}
```



# IntelliJ: creazione dei test

Possiamo velocizzare la scrittura dei test?



# IntelliJ: creazione dei test

ES: scriviamo una semplice classe Divider:

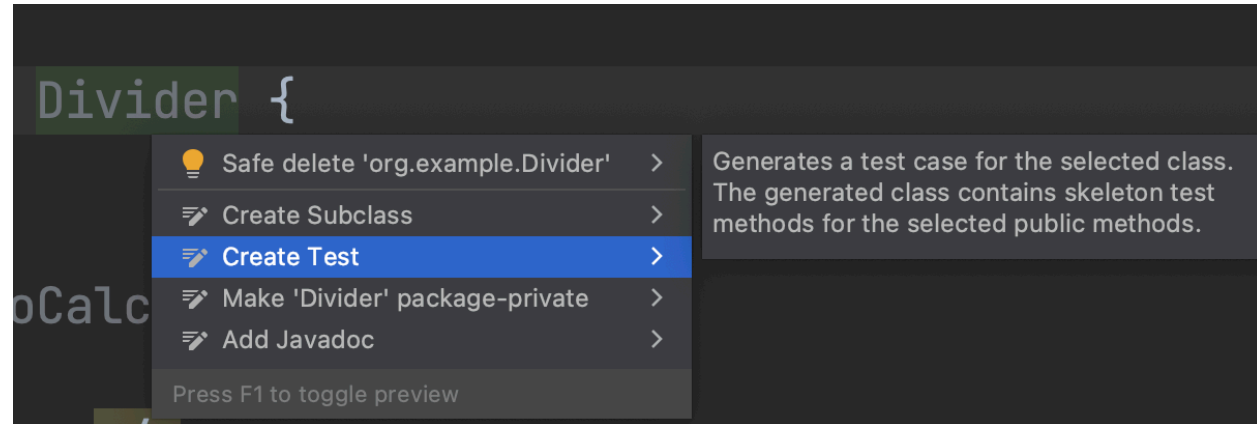
```
public class Divider {  
    double doCalc(int x, int y){  
        return x/y;  
    }  
}
```



# IntelliJ: creazione dei test

IntelliJ ci "aiuta":

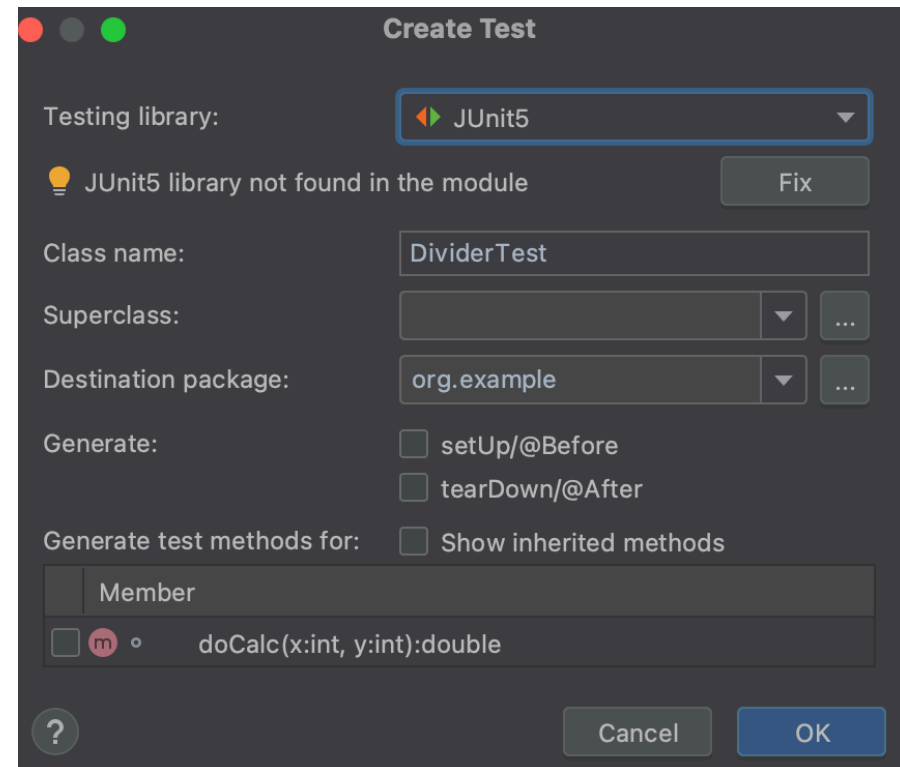
Alt Invio



JUnit 5.

Scegliete quali metodi testare.

*Se appare Fix. ok!*



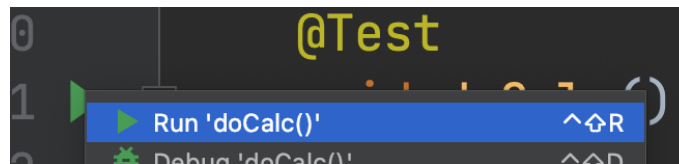


# IntelliJ: creazione dei test

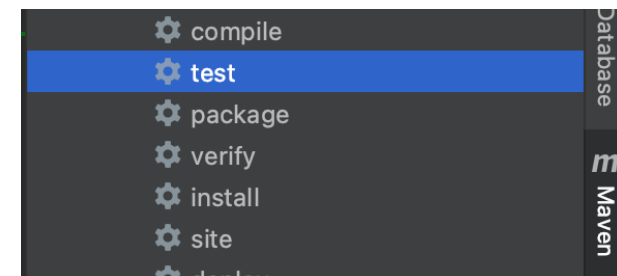
modifichiamo:

```
class DividerTest {  
  
    @Test  
    void doCalc() {  
  
        Divider divider = new Divider();  
        double r = divider.doCalc(3, 2);  
        assertEquals(1.5, r);  
  
    }  
}
```

run...



Or better:





# IntelliJ: creazione dei test

Perche' non funziona?

Git repo:  
<https://github.com/ingconti/JUnitIntro>

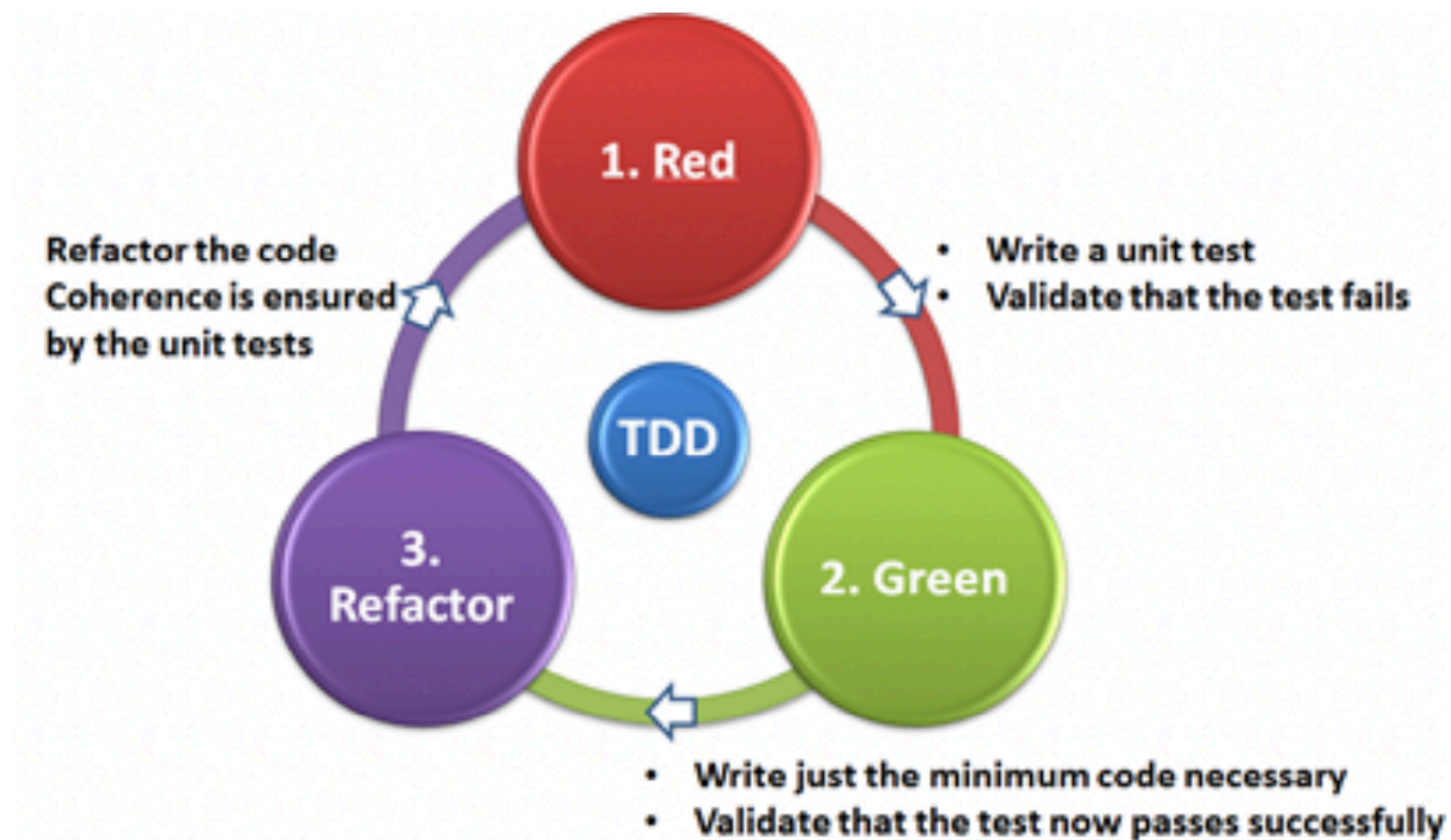
Esempio più complesso (da anno precedente...)

Logica:

- classe LoginManager (tralasciamo x ora singleton etc..)
- Array di nickname
- validiamo doppio inserimento (per ora solo nome..)

(useremo un approccio TDD..)

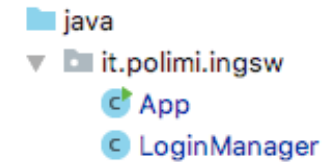




```
public class LoginManager {  
}
```

- aggiungiamo:

```
public class LoginManager {  
    ArrayList<String> nicknames;  
  
    void allocateLazy(){  
        if (nicknames == null) {  
            nicknames = new ArrayList<String>();  
        }  
    }  
  
    Boolean login(String nick ) {  
        allocateLazy();  
        int count = nicknames.size();  
        if (count >= 4)  
            return false;  
  
        nicknames.add(nick);  
        return true;  
    }  
}  
  
(serve import java.util.ArrayList;)
```



il test:

```
import static org.junit.jupiter.api.Assertions.*;  
import org.junit.jupiter.api.Test;
```

```
class AppTest{  
  
    @Test  
    void LoginTest() {  
        System.out.println("testing");  
        LoginManager lm = new LoginManager();  
        assertEquals(true, lm.login("bob"));  
        assertEquals(false, lm.login("bob"));  
    }  
}
```

il test: bob NON puo' loggare 2 volte..

run..

```
org.opentest4j.AssertionFailedError:
```

```
Expected :false
```

```
Actual   :true
```

WHY?

TDD ha fatto suo dovere:

deve essere:

il metodo login e' errato:

```
Boolean login(String nick) {  
    allocateLazy();  
    int count = nicknames.size();  
    if (count >= 4)  
        return false;  
  
    nicknames.add(nick);  
    return true;  
}
```

```
Boolean login(String nick) {  
    allocateLazy();  
    int count = nicknames.size();  
    if (count >= 4)  
        return false;  
  
    → if (nicknames.contains(nick))  
        return false;  
  
    nicknames.add(nick);  
    return true;  
}
```

# Alcuni consigli

- Una cosa per volta e per metodo test
  - Dieci piccoli test sono molto meglio di un solo test dieci volte più grande
- Ogni metodo test dovrebbe avere pochi (un) metodo assert
  - La prima assert che fallisce blocca il test
  - Non si riuscirebbe quindi a sapere se eventuali asserzioni successive fallirebbero
- I test dovrebbero evitare logiche complesse
  - Minimizzare l'uso di if, loops, switch, ecc
  - Evitare try/catch
    - Se l'eccezione fosse sollevabile, bisognerebbe usare expected
    - Altrimenti sarebbe meglio lasciare che JUnit gestisca l'eccezione
- Test complessi vanno bene, ma solo in aggiunta a quelli semplici

# “Bad smells”

- Ogni test dovrebbe essere auto-contenuto e non considerare gli altri
- Cose da evitare
  - Vincolare l'ordine di applicazione dei test
  - I test si chiamano a vicenda
  - Uso di oggetti condivisi

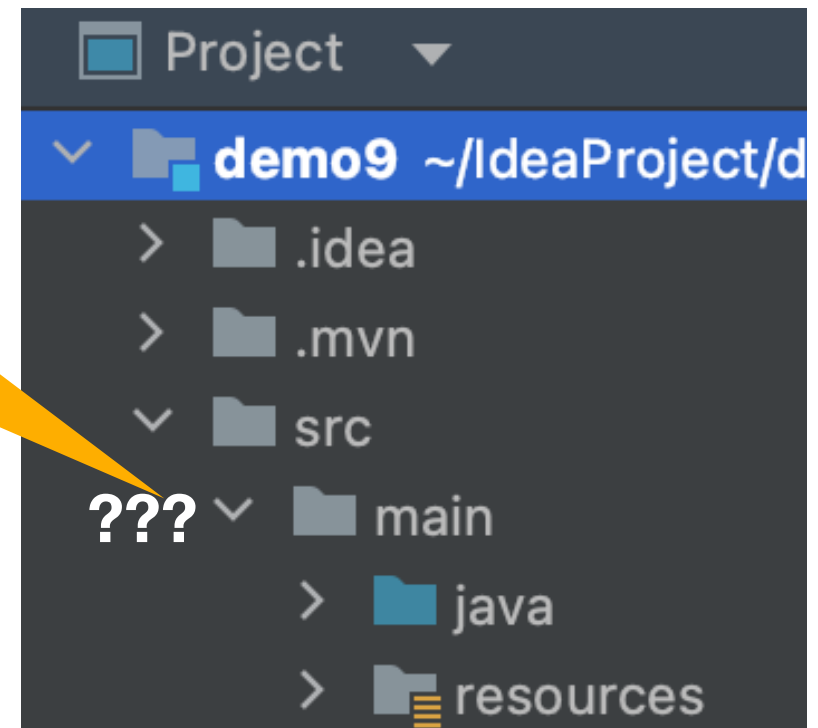
# Programmazione difensiva

- Un pizzico di paranoia può essere utile
- Possiamo/dobbiamo scrivere i programmi in modo che scoprano e gestiscano ogni possibile situazione anomala:
  - procedure chiamate con parametri attuali scorretti,
  - file: devono essere aperti ma sono chiusi, devono aprirsi e non si aprono...
  - riferimenti ad oggetti null, array vuoti ...
- Il meccanismo delle eccezioni è un aiuto utile
- Essere scrupolosi con il test
  - ricordarsi che l'obiettivo è trovare gli errori, non essere contenti di non trovarne
- Può convenire dare ad altri il compito di collaudare i propri programmi



# WAIT...

Ho fatto un project  
intellij/Javafx  
e NON ho i test!!!

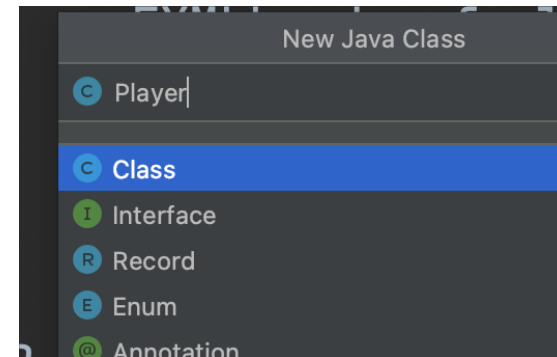


# Test for IntelliJ/JavaFx project

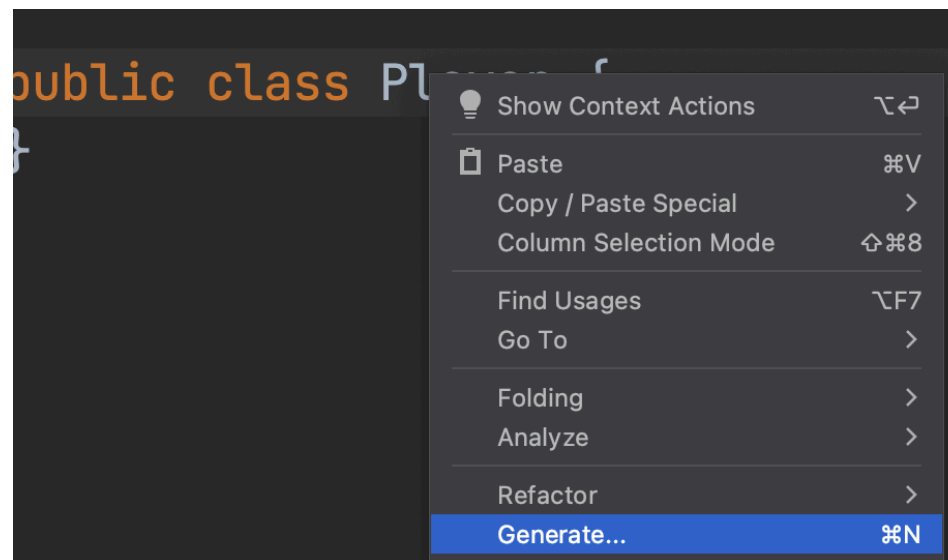
Nessun problema! Al primo test create vi apparirà tutto...

Creiamo una classe (una delle vostre...)

Es. Player:

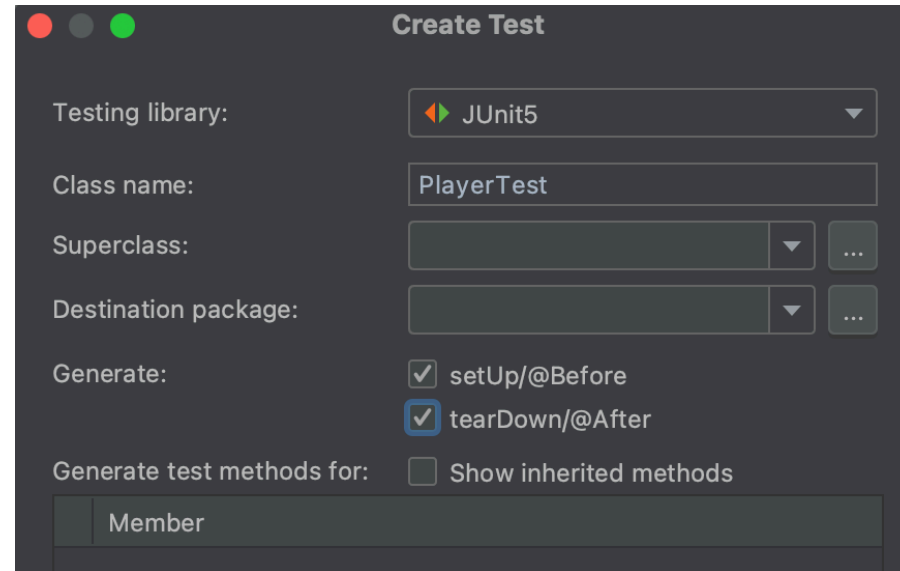


Right click...



# Test for IntelliJ/JavaFx project

..



....

**Mission accomplished!**

