# JUnit – unit testovanie v Jave

Mýliť sa je ľudské. Aj najskúsenejší programátor občas napíše chybu, ktorá na prvý pohľad vyzerá nevinná, no pri nasadení môže spôsobiť veľké problémy. Predstav si softvér ako vlakovú dopravu: jedna chyba môže spôsobiť iba oneskorenie, alebo, v tom horšom prípade, kolíziu dvoch vlakov. Preto je dôležité chyby odhaliť skôr, než spôsobia nejakú škodu.

Unit testovanie je jednoduchý a efektívny spôsob, ako to dosiahnuť. Umožní ti izolovať malé časti kódu a overiť, že robia presne to, čo majú. Namiesto zdĺhavého hľadania v stovkách riadkov kódu rýchlo nájdeš problém, opravíš ho a vieš, že oprava nič iné nerozbila. Unit testy sú ako kontrolné body v továrni alebo senzory v aute, overia očakávané správanie a včas upozornia na chyby.

**JUnit** je najrozšírenejší framework pre unit testy v Jave. Pomôže ti písať krátke, rýchle a opakovateľné testy, ideálne pri refaktoringu, pridávaní funkcií alebo nasadzovaní do produkcie. V článku nájdeš praktický a použiteľný sprievodca, ktorý ťa prevedie krokmi:

* čo je unit testovanie a prečo mu venovať čas,
* predstavenie JUnit (vrátane JUnit 5) a jeho výhod,
* krok-za-krokom návod, ako vytvoriť a spustiť testy v IntelliJ IDEA,
* prehľad najdôležitejších anotácií a ukážky kódu, ktoré môžeš použiť hneď,
* overené tipy a triky, ktoré ti ušetria hodiny práce.

Či píšeš svoj prvý test, alebo chceš vylepšiť stabilitu veľkého projektu, tento článok ti poskytne konkrétne návody a nástroje, ktoré môžeš hneď využiť. Začneme tým, čo unit testovanie znamená v praxi a prečo sa táto investícia oplatí.

## Čo je unit testovanie?

Unit testovanie je spôsob, ako si vývojár overí, že najmenšie časti programu fungujú presne tak, ako majú. Tieto časti sa nazývajú „jednotky“ (units) a väčšinou ide o jednotlivé metódy alebo triedy. Pointa je jednoduchá. Ak vieš, že jednotlivé základné stavebné bloky aplikácie sú správne, máš oveľa väčšiu istotu, že sa spoľahlivo správa aj celok.

Pri unit testovaní píšeš špeciálne testovacie metódy, ktoré automaticky spustia kód s konkrétnymi vstupmi a skontrolujú, či výsledok zodpovedá očakávaniu. Tieto testy bežia rýchlo, bez závislosti od databáz či externých služieb, aby bolo možné ich spúšťať opakovane a okamžite získať spätnú väzbu.

Dôležitou vlastnosťou unit testov je izolácia. Každý test sa sústreďuje na presne jednu funkcionalitu a ignoruje všetko ostatné. Takto je jednoduchšie odhaliť zdroj problému. Ak test zlyhá, vieš presne, ktorá časť kódu nefunguje. V praxi to znamená, že namiesto manuálneho testovania kliknutím v aplikácii sa spoliehaš na automatizované testy, ktoré dokážeš spustiť kedykoľvek počas vývoja. Výsledok? Menej chýb prenikne do vyšších vrstiev systému, rýchlejšie sa odhaľujú regresie a vývojári majú väčšiu dôveru realizovať refaktoring a rozširovanie aplikácie. Unit testovanie teda nie je len kontrola správnosti kódu, ale aj nástroj, ktorý prispieva k čitateľnosti, udržiavateľnosti a dlhodobej kvalite softvéru.

## Prečo písať unit testy?

Na prvý pohľad sa môže zdať, že písanie unit testov je práca navyše. Veď aplikácia sa predsa rozbehne aj bez nich. Skúsenosť však ukazuje, že práve testy sú to, čo oddeľuje krátkodobý úspech od dlhodobo udržateľného vývoja.

Unit testy prinášajú niekoľko kľúčových výhod:

**Rýchle odhalenie chýb**  
Čím skôr sa chyba nájde, tým lacnejšie je jej opravenie. Unit testy odhalia problémy už vo fáze vývoja, ešte predtým, ako sa kód dostane do produkcie.

**Istota pri zmenách a refaktoringu**  
Každý projekt sa vyvíja a mení. Unit testy fungujú ako bezpečnostná sieť. Ak po úpravách niečo prestane fungovať, testy to okamžite odhalia.

**Lepšia čitateľnosť a dizajn kódu**  
Keď píšeš testy, prirodzene ťa to núti rozdeliť kód na menšie a zrozumiteľnejšie časti. Testovateľný kód je často aj čistejší kód.

**Automatizácia namiesto manuálnej kontroly**  
Namiesto opakovaného klikania v aplikácii či spúšťania scenárov ručne stačí pustiť testy. Ušetríš čas a získaš konzistentné výsledky.

**Vyššia dôvera tímu a zákazníka**  
Unit testy budujú istotu, že produkt je stabilný. Tím sa môže viac sústrediť na nové funkcionality a zákazník získava dôveru v kvalitu riešenia.

Unit testy nie sú len o odhaľovaní chýb. Sú investíciou do budúcnosti projektu, lebo znižujú riziko, zlepšujú kvalitu a zrýchľujú tempo vývoja. Vďaka nim sa vývojári menej obávajú meniť kód a môžu pracovať sebavedomejšie aj na komplexných úlohách.

## Čo je JUnit? Prehľad JUnit 5

JUnit je open-source framework, ktorý sa stal štandardom pre unit testovanie v Jave. Vznikol ešte koncom 90. rokov a odvtedy sa používa v tisíckach projektov po celom svete. Jeho hlavná úloha je jednoduchá a to poskytnúť vývojárom nástroje na písanie, organizovanie a spúšťanie automatických testov priamo v Jave.

### Prečo práve JUnit?

* Je ľahko dostupný a bezplatný, takže ho môže použiť každý Java vývojár.
* Je integrovaný do populárnych IDE (IntelliJ IDEA, Eclipse, NetBeans), vďaka čomu spustíš testy jediným kliknutím alebo klávesovou skratkou.
* Dá sa jednoducho prepojiť s build nástrojmi ako Maven či Gradle, čo umožňuje spúšťať testy automaticky pri každom buildovaní projektu.
* Je flexibilný a rozšíriteľný, vďaka čomu sa dá prispôsobiť rôznym projektom a potrebám.

### Evolúcia až k JUnit 5

Predchádzajúce verzie (JUnit 3 a 4) definovali základné princípy testovania v Jave, no mali aj svoje limity. JUnit 5 priniesol zásadnú zmenu architektúry a rozdelil framework na tri časti:

**JUnit Platform** - základná vrstva, ktorá sa stará o spúšťanie testov a integráciu s IDE či buildovacími nástrojmi.

**JUnit Jupiter** - poskytuje nové API a anotácie pre písanie testov (napr. @Test, @BeforeEach, @AfterEach). Toto je hlavná časť, s ktorou vývojár pracuje.

**JUnit Vintage** - umožňuje spúšťať staršie testy napísané v JUnit 3 a 4, aby bola zabezpečená spätná kompatibilita.

### JUnit 5 prehľad

[JUnit 5](https://junit.org) priniesol navyše:

* Nové anotácie a flexibilnejší lifecycle testov (napr. *@BeforeAll, @AfterAll, @Disabled, @ParameterizedTest*).
* Podporu parameterizovaných testov, ktoré umožňujú spúšťať rovnaký test s viacerými vstupmi.
* Rozšíriteľnosť cez Extension API, vďaka ktorej si vieš doplniť vlastné správanie testov (napr. logovanie, mockovanie, integráciu s databázou).
* Lepšiu integrácia s modernými nástrojmi a podpora pre Java 8+ (vrátane lambda výrazov a stream API).

JUnit 5 je moderný a výkonný nástroj, ktorý si zachováva jednoduchosť starších verzií, no zároveň ponúka flexibilitu potrebnú pre dnešné komplexné aplikácie. Ak sa chceš naučiť testovať v Jave efektívne a podľa aktuálnych štandardov, JUnit 5 je voľba číslo jeden.

**Používateľská príručka:** <https://docs.junit.org/current/user-guide/>  
**JUnit API:** <https://docs.junit.org/current/api/>

***Vedel si že ... ?***  
Vývojári JUnit frameworku nelenia a už pripravujú novú verziu. JUnit 6 je momentálne v aktívnom vývoji a jeho vydanie je naplánované na koniec roku 2025.

## Ako funguje JUnit testovanie

Aby sme pochopili užitočnosť JUnit, je dôležité vedieť, ako celý proces testovania prebieha. JUnit sa stará o to, aby každý test prebehol v kontrolovanom prostredí, ktoré je izolované od ostatných testov. V praxi to znamená, že testy sú predvídateľné, nezávislé a opakovateľné.

### Spustenie testov

Celý proces sa začína spustením testovacej triedy. O to sa môže postarať buď IDE (napr. IntelliJ IDEA alebo Eclipse), build nástroje ako Maven a Gradle, alebo automatizované pipeline v CI/CD. JUnit načíta triedu, vyhľadá v nej testovacie metódy a pripraví všetko potrebné na ich vykonanie.

### Izolované prostredie

Každý test by mal byť nezávislý od ostatných. JUnit preto pre každý test vytvorí novú inštanciu testovacej triedy. To zabezpečí, že výsledky testu nie sú ovplyvnené predchádzajúcim spustením a každý test začína s čistým stavom.

### Lifecycle testu

Pri vykonaní testov JUnit dodržiava presný životný cyklus. Najskôr sa pripravia spoločné zdroje, potom sa inicializuje prostredie pre konkrétny test, následne sa spustí samotná testovacia metóda a napokon sa prostredie vyčistí. Tento prístup zabraňuje tomu, aby sa testy navzájom ovplyvňovali a zaručuje konzistentnosť výsledkov.

### Overenie výsledku

Každý test je v podstate malý experiment, očakávame určitý výsledok a JUnit overí, či sme ho dosiahli. Framework automaticky zaznamenáva úspech alebo zlyhanie testu a poskytuje spätnú väzbu. Ak test zlyhá, vývojár okamžite vie, ktorá časť kódu sa nespráva podľa očakávania.

### Výstup testovania

Po dokončení testov JUnit vyhodnotí výsledky a zobrazí ich prehľadne: úspešné testy sú zelené, zlyhané testy červené, preskočené testy majú špeciálne označenie. Výstup testovania vývojárovi okamžite identifikovať problémové oblasti. V kombinácii s detailným logom testov ide o veľmi efektívny nástroj na diagnostiku chýb.

### Automatizácia a integrácia

JUnit testovanie sa stáva najefektívnejším vtedy, keď sa integruje do automatizovaného buildovacieho procesu. Pri každej zmene v kóde sa testy spustia automaticky, čím sa predchádza tomu, že sa do projektu dostane chyba. To výrazne zvyšuje spoľahlivosť a rýchlosť vývoja, pretože problémy sa odhaľujú včas, nie až po nasadení aplikácie.

## JUnit anotácie

Jednou z najväčších výhod JUnit je jeho jednoduchý a konzistentný spôsob práce s testami, ktorý je postavený práve na anotáciách. Vďaka nim nemusíš písať zložitú infraštruktúru, stačí pridať anotáciu a JUnit sa postará o všetko ostatné: od spúšťania testov, cez prípravu dát až po vyhodnotenie výsledkov.

Anotácie určujú kedy a ako sa testy vykonajú, umožňujú ich organizovať, opakovať alebo parametrizovať a zjednodušujú prácu aj pri väčších projektoch. Pre vývojára sú preto kľúčovým nástrojom, ktorý robí testovanie rýchlejším, spoľahlivejším a čitateľnejším.

### Najdôležitejšie JUnit 5 anotácie

|  |  |  |
| --- | --- | --- |
| Anotácia | Popis | Použitie |
| @Test | Označuje testovaciu metódu. | Základ každého unit testu. |
| @BeforeEach | Spustí sa pred každým testom. | Inicializácia objektov, reset stavov. |
| @AfterEach | Spustí sa po každom teste. | Uvoľnenie zdrojov, čistenie. |
| @BeforeAll | Spustí sa raz pred všetkými testami v triede. | Nastavenie databáz, konfigurácia. |
| @AfterAll | Spustí sa raz po všetkých testoch v triede. | Zatvorenie spojení, cleanup. |
| @Disabled | Dočasne preskočí test alebo celú triedu. | Pri nedokončených alebo nefunkčných testoch. |
| @DisplayName | Nastaví čitateľný názov testu. | Zlepšuje reporty v IDE alebo CI/CD. |
| @ParameterizedTest | Umožní spustiť ten istý test s viacerými vstupmi. | Efektívne testovanie rôznych dát. |
| @RepeatedTest | Spustí test viackrát. | Testovanie nestabilných/náhodných procesov. |
| @Tag | Označí test kategóriou. | Filtrovanie testov (napr. fast/slow). |
| @Nested | Vnorené triedy pre organizáciu testov. | Logické zoskupenie testov. |
| @Timeout | Určí maximálny čas spustenia testu. | Prevencia zaseknutých testov. |
| @ExtendWith | Rozšíri testy o externé rozšírenia. | Integrácia (napr. Spring, Mockito). |

## Vytvorenie a spustenie prvého JUnit testu v IntelliJ IDEA (krok za krokom)

V IntelliJ IDEA je práca s JUnit testami veľmi pohodlná. IDE poskytuje integrovanú podporu pre JUnit 5, takže vytváranie a spúšťanie testov je rýchle a jednoduché, aj pre začiatočníkov. Nasledujúci postup ti ukáže, ako vytvoriť svoj prvý test a okamžite ho spustiť.

Predpokladom pre nasledovanie tohto tutoriálu je mať nainštalované vývojové (IDE) prostredie IntelliJ IDEA a JDK (Java Development Kit). Ak ho nemáš, tak sme pre teba vytvorili aj [tutoriál aj na inštaláciu vývojového prostredia](https://msgprogramator.sk/java-intellij-idea/).

### 1. Vytvorenie projektu alebo modulu

Ak ešte nemáš projekt, vytvor nový Java projekt:

Vyber **File** → **New** → **Project**.

Zvoľ **Java** a klikni na **Next**.

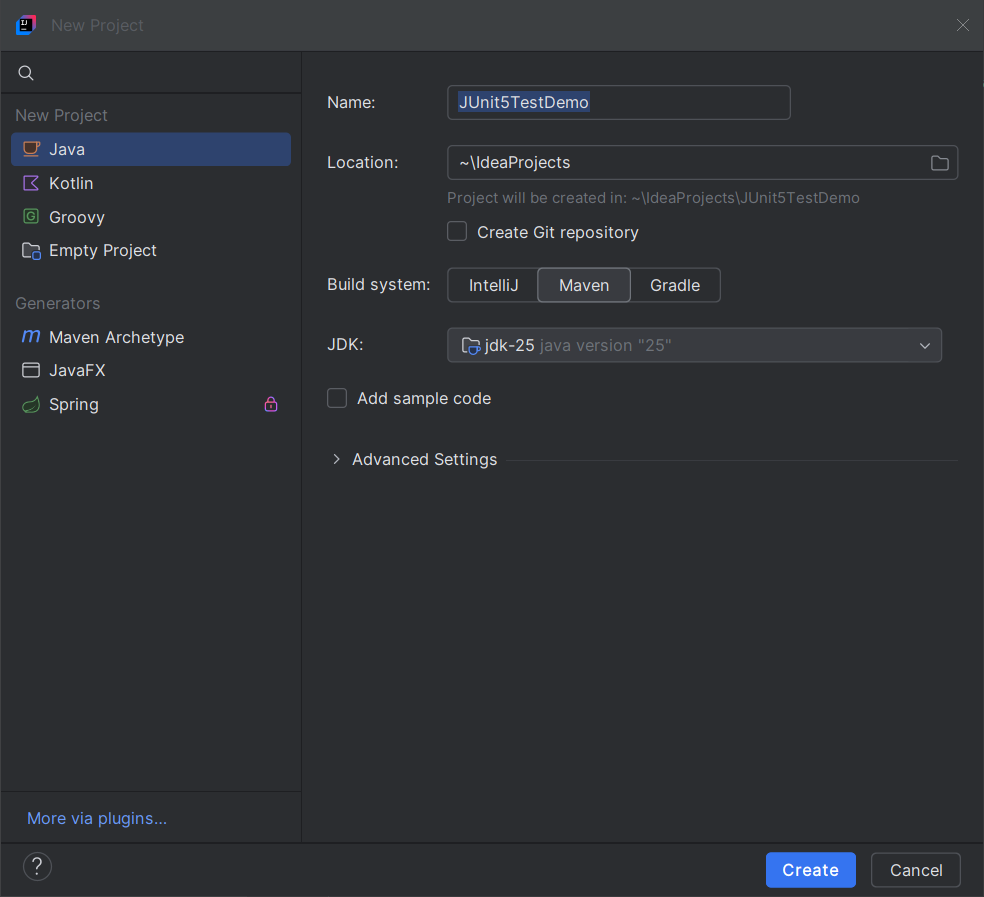
Zadaj názov projektu (napr. *JUnit5TestDemo*) a umiestnenie projektu.

Ako Build system vyber **Maven** alebo **Gradle**.

Skontroluj, že máš pridanú JDK (Java Development Kit).

Ak máš už existujúci projekt, môžeš pokračovať priamo k pridaniu testovacieho súboru.

**Odporúčanie:** Pre moderné projekty sa oplatí používať **Maven alebo Gradle**, pretože spravujú všetky závislosti automaticky a umožňujú jednoduchú integráciu testov do CI/CD. Manuálne pridávanie JAR knižníc sa hodí len na malé alebo jednoduché projekty.



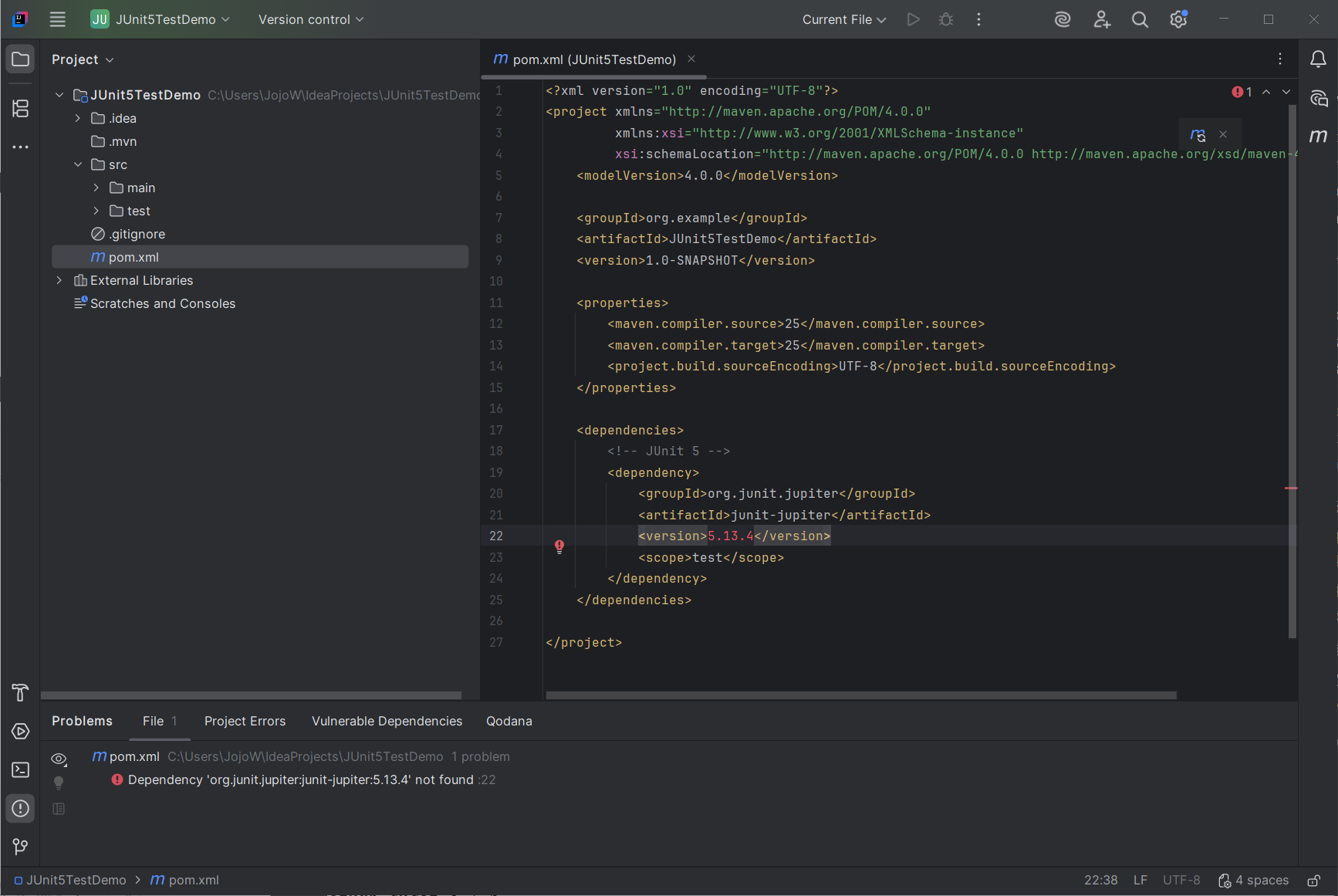
### 2. Pridanie JUnit 5 do projektu

Pre správnu funkciu testov musí byť JUnit 5 zahrnutý v závislostiach projektu.  
Ak používaš **Maven**: pridaj do pom.xml závislosť pod sekciu properties:

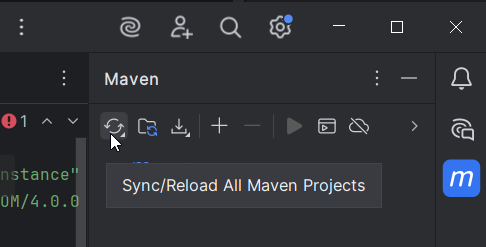


Ak používaš **Gradle**: pridaj do build.gradle:



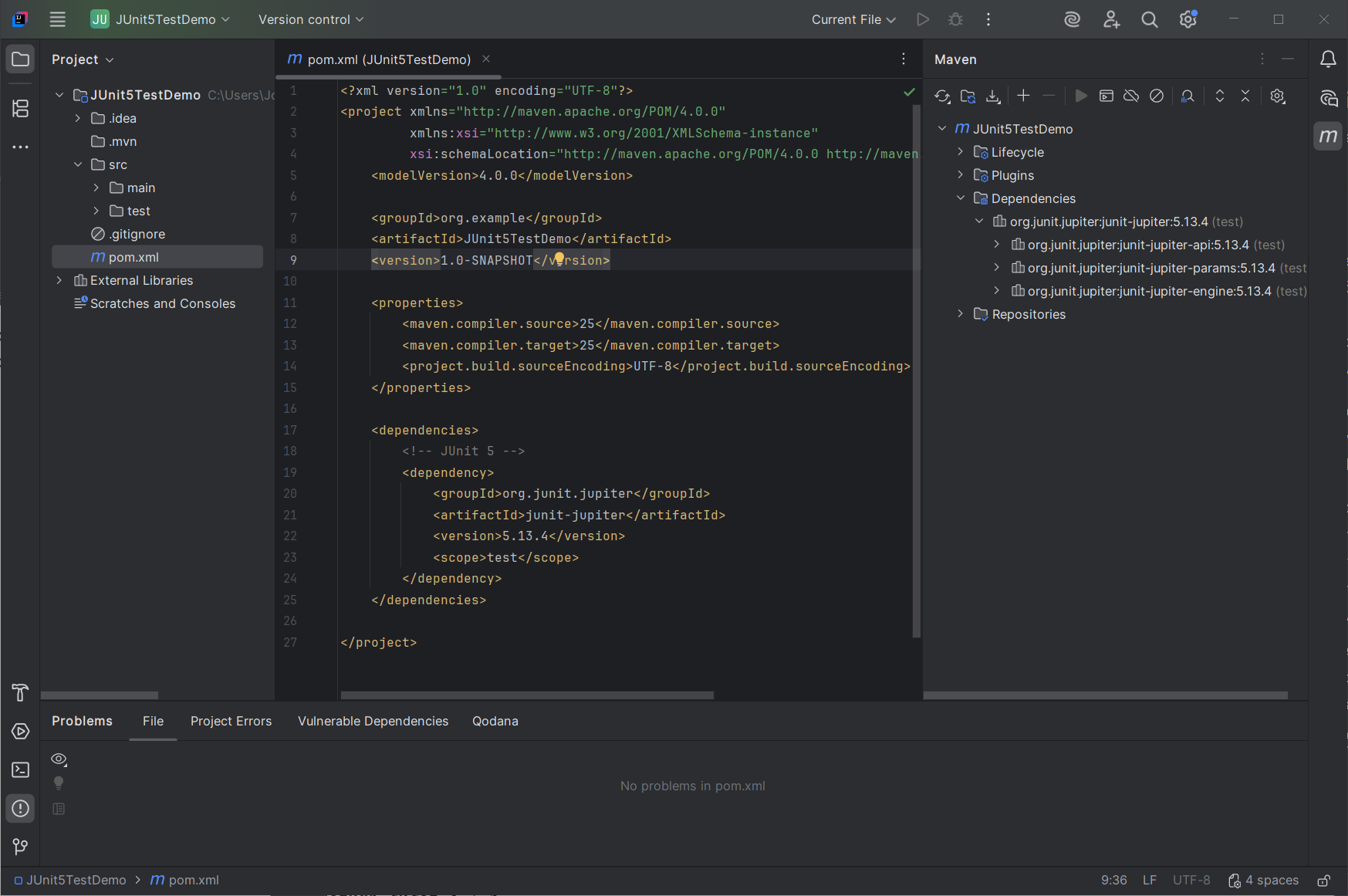


V našom projekte použijeme buildovací systém Maven. Po upravení súboru pom.xml a pridaní závislosti (dependency) na JUnit5, môžeme dostať chybu, že závislosť sa nenašla. Aby sme chybu opravili, v pravej časti si klikneme na ikonku Mavenu a stačíme prvé tlačidlo pre synchronizáciu a opätovné načítanie všetkých Maven projektov.



IntelliJ následne automaticky stiahne knižnicu a umožní ti používať anotácie JUnit 5. To že sa to podarilo sa dá verifikovať nasledovne:

1. Zmizla predchádzajúca červená chyba z pom.xml na neexistujúcu závislosť.
2. V Maven okne uvidíme všetky pridané závislosti.



Všimnime si na obrázku, že v prípade Maven projektu máme v stromovej štruktúre projektu ***src*** dva oddelené priečinky: ***main*** a ***test***. Do priečinku ***main*** pridávame implementáciu biznis logiky programu, to znamená náš Java kód a do priečinku ***test*** všetky potrebné testy na otestovanie napísaného kódu. Takto zostávajú kód programu a samotné testy prehľadne oddelené.

### 3. Vytvorenie biznis logiky programu

Aby sme mali neskôr čo otestovať potrebujeme si napísať nejaký program. Ja som si pre teba pripravil výpočet magického čísla 6174 – Kaprekarovej konštanty.

#### Kaprekarova konštanta

Kaprekarova konštanta je číslo 6174. Ide o štvormiestne číslo objavené indickým matematikom D. R. Kaprekarom. Možno sa teraz pýtaš, prečo to označujeme ako magické číslo?

Zvolíš ľubovoľné štvormiestne číslo, ktoré má aspoň dve rôzne cifry, z týchto cifier vytvoríš najväčšie a najmenšie možné číslo a od väčšieho odčítaš menšie. Opakovaním tejto operácie **sa vždy dostaneš na 6174 najviac do siedmich krokov**. Tento postup, ale neplatí pre zvolené číslo obsahujúce všetky rovnaké cifry (napr. 1111), potom výsledok bude 0.

#### Postup výpočtu Kaprekarovej konštanty

1. Vyber štvormiestne číslo: musí mať štyri cifry a nesmie obsahovať všetky rovnaké cifry (napr. 1111 alebo 2222 sú neprípustné).
2. Zoraď cifry do najväčšieho čísla: poskladaj cifry tak, aby vzniklo čo najväčšie možné štvormiestne číslo.
3. Zoraď cifry do najmenšieho čísla: rovnaké cifry poskladaj tak, aby vzniklo čo najmenšie možné číslo (nezabudni doplniť nuly, ak treba, napr. 0523 → 523 → formátuj ako 0523).
4. Odčítaj: od väčšieho čísla odčítaj menšie.
5. Opakuj: výsledok použiješ ako nové vstupné číslo a kroky 2–4 opakuješ, až kým nedostaneš 6174 (alebo 0, ak pôvodné číslo malo všetky cifry rovnaké).

Príklad výpočtu si môžeš pozrieť v priloženom videu.

[](https://www.youtube.com/embed/xtyNuOikdE4?feature=oembed)

V priečinku ***main*** si vytvoríme triedu *KaprekarRoutine* a vložíme do nej nasledovný Java kód, pre ktorý si v nasledujúcom kroku napíšeme jednotkové testy pomocou JUnit5 frameworku.



Trieda **KaprekarRoutine** implementuje výpočet, koľko krokov je potrebných, aby sa ľubovoľné štvormiestne číslo (s aspoň dvomi rôznymi ciframi) premenilo na **Kaprekarovu konštantu 6174**.

* Metóda **stepsToKaprekar(int number)**:  
  Skontroluje, či je číslo platné (musí mať 4 cifry a nesmú byť všetky rovnaké). Potom opakovane vykonáva tzv. Kaprekarov krok, až kým nedosiahne 6174. Počíta pritom počet krokov a vráti ho. Ak by krokov bolo viac než 7 (čo by sa v praxi nemalo stať), vyhodí chybu.
* Metóda **kaprekarStep(int number)**:  
  Zoberie číslo, zoradí jeho cifry vzostupne aj zostupne, vytvorí z nich najmenšie a najväčšie možné číslo a vráti ich rozdiel.
* Metóda **allDigitsSame(int number)**:  
  Skontroluje, či všetky cifry čísla sú rovnaké.

### 4. Otestovanie programu (JUnit5)

Ak máme hotovú biznis logiku programu, potrebujeme si dôkladne otestovať, že kód funguje presne podľa špecifikácie. V našom prípade sa zameriame na testovanie platných a neplatných čísiel (vstupov) a hraničných prípadov.

V priečinku ***test*** si vytvoríme triedu *KaprekarRoutineTest* a vložíme do nej nasledovný Java kód:



### Zhrnutie testov

Testy platnosti čísel (Valid numbers) ***testAlreadyKaprekarConstant***: číslo 6174 má vrátiť 0 krokov.  
***testNumber3524***: číslo 3524 má dosiahnuť 6174 v 3 krokoch.  
***testNumber2111***: číslo 2111 má dosiahnuť 6174 v 5 krokoch.  
***testNumber1234Repeated***: číslo 1234 opakovane overuje, že sa výsledok vždy dostane na 6174 v ≤ 7 krokoch.

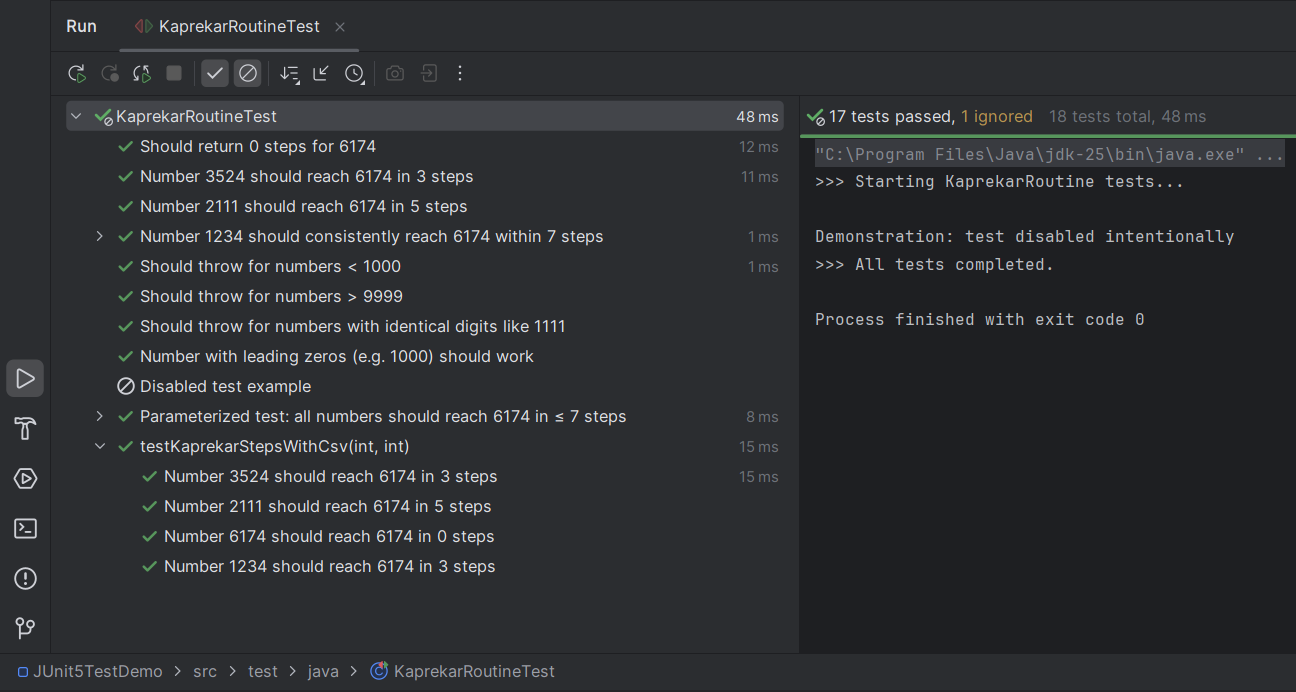
Testy neplatnosti čísel (Invalid inputs) ***testTooSmallNumber***: číslo < 1000 vyhodí IllegalArgumentException.  
***testTooLargeNumber***: číslo > 9999 vyhodí IllegalArgumentException.  
***testAllDigitsSame***: číslo so všetkými rovnakými ciframi (1111) vyhodí IllegalArgumentException.

Hraničné prípady (Edge cases) ***testNumberWithLeadingZeros***: číslo s nulami (napr. 1000) správne funguje a dostane sa na 6174. ***disabledTest***: ukážka testu, ktorý je zakázaný (@Disabled). ***testKaprekarWithinSevenSteps***: parameterizovaný test, ktorý overuje viac vstupov naraz (3524, 2111, 1234, 1000), všetky musia dosiahnuť 6174 najneskôr do 7 krokov.  
***testKaprekarStepsWithCsv***: parameterizovaný test, ktorý pre viaceré vstupy (3524, 2111, 6174, 1234) overuje, že metóda stepsToKaprekar vráti presne očakávaný počet krokov (3, 5, 0, 3).

**Použili sme nasledovné JUnit 5 anotácie:**

* @Test – základný test.
* @RepeatedTest – opakovanie testu viackrát.
* @ParameterizedTest + @ValueSource – testovanie viacerých vstupov jednou metódou.
* @ParameterizedTest + @CsvSource – testovanie viacerých vstupov a výstupov jednou metódou.
* @BeforeAll, @AfterAll, @BeforeEach, @AfterEach – životný cyklus testov.
* @Order + @TestMethodOrder – kontrola poradia spúšťania testov.
* @DisplayName – čitateľnejší názov testu.
* @Tag – označenie testu kategóriou (napr. edge cases).
* @Disabled – vypnutý test.

**Výstup testov môže vyzerať nasledovne**



Pripravili sme pre teba súbory so spomínaným príkladom vo forme kódu, ktorý si môžeš spustiť priamo v Jave. Stiahni si Java kód pre **JUnit5TestDemo** tu.

## Java unit testy a JUnit – tipy a triky na písanie testov

Testovanie je kritickou súčasťou vývoja softvéru, pretože nám umožňuje overiť, či jednotlivé časti nášho kódu fungujú správne a spĺňajú požiadavky. Unit testovanie je jednou z najdôležitejších metód testovania, ktorá sa zaoberá testovaním jednotlivých častí kódu, ako sú metódy a triedy, oddelene od zvyšku aplikácie.

V Jave je testovanie jednotlivých komponentov bežnou praxou, avšak písanie efektívnych a spoľahlivých unit testov si vyžaduje isté znalosti a zručnosti. V tomto článku sa pozrieme na niektoré užitočné tipy a triky, ktoré ti pomôžu napísať kvalitné unit testy.

### Používaj JUnit knižnicu

**JUnit** je najpopulárnejšia testovacia knižnica pre jazyk Java, ktorá poskytuje prostredie pre písanie a spúšťanie unit testov. Umožňuje nám definovať testovacie prípady pomocou anotácií a poskytuje množstvo metód pre overenie očakávaných výsledkov. Pre začiatok je potrebné integrovať JUnit do tvojho projektu ako závislosť.

### Oddeľ testy od hlavného zdrojového kódu

Testovacie triedy by si mal držať oddelené od hlavného zdrojového kódu, aby sa zabránilo možnosti ich spustenia v produkčnom prostredí. Najlepšie miesto pre testy bude relatívna cesta src/main/test a práve to je aj adresár, kde buildovacie nástroje ako napríklad Maven budú hľadať implementované testy.

### Pomenuj testy výstižnými názvami

Pri písaní unit testov je dôležité pomenovať testovacie metódy jednoznačným a výstižným spôsobom. Testovacie metódy by mali začínať slovom „test“ a mali by jasne popisovať, čo sa daný test snaží overiť. To pomáha pri rýchlej identifikácii problémov, ak niektorý z testov zlyhá.

### Začni s jednoduchými testami

Pri začiatku písania testov sa sústreď na jednoduché prípady, ktoré pokrývajú základné funkcie tvojich tried. Týmto spôsobom sa presvedčíš, že základné komponenty tvojej aplikácie fungujú správne, a potom môžeš postupne rozširovať testovacie scenáre.

### Testuj hraničné hodnoty

Hraničné hodnoty sú častým zdrojom chýb v kóde. Pri písaní testov sa uisti, že zahrňuješ hraničné hodnoty, ktoré môžu ovplyvniť správanie tvojho kódu. Toto zahŕňa neinicializované hodnoty objektov (**null**), záporné hodnoty, maximálne a minimálne hodnoty, a tiež neplatné vstupy. Otestovanie týchto hraničných hodnôt môže odhaliť potenciálne chyby v kóde.

### Izoluj testy od vonkajších závislostí

Unit testy by mali byť nezávislé a nemali by závisieť od vonkajších zdrojov, ako sú databázy, súbory alebo webové služby. Na dosiahnutie tohto cieľa môžeš použiť mockovanie alebo stubovanie, ktoré simulujú správanie týchto závislostí. Takto zabezpečíš, že zlyhanie testu nebude spôsobené problémami s externými závislosťami.

### Využívaj assert metódy

JUnit knižnica poskytuje rôzne *assert* metódy , ktoré umožňujú overiť, či je očakávaný výstup tvojho testu správny. Medzi najpoužívanejšie patrí *assertEquals()*, *assertTrue()*, *assertFalse()*, atď. Použitie správnych *assert* metód je kľúčové pre úspešné overovanie správnosti testov.

### Píš parametrizované testy

JUnit podporuje parametrizované testy, ktoré ti umožňujú spúšťať rovnaké testy s rôznymi vstupnými hodnotami. Týmto spôsobom môžeš zjednodušiť testovanie rôznych scenárov a minimalizovať duplicity v testovacom kóde.

### Pravidelne spúšťaj testy

Nezabudni pravidelne spúšťať všetky unit testy. Často sa stáva, že zmeny v jednej časti kódu môžu ovplyvniť inú časť aplikácie. Pravidelné spúšťanie testov ti umožní zachytiť problémy skôr, než sa dostanú do produkčného prostredia.

### Testuj vyvolanie výnimiek

Nevhodné ošetrenie výnimiek môže spôsobiť pád tvojho programu. Preto otestuj, či tvoje metódy správne vyhodnocujú očakávané výnimky a správne na tieto výnimky dokážu zareagovať.

V JUnit môžete použiť anotáciu @Test spolu s parametrom expected na overenie, či metóda vyvolá určitú výnimku. Napríklad: *@Test(expected = IllegalArgumentException.class)*

### Analyzuj pokrytie kódu

Test coverage (pokrytie testami) je metrika, ktorá ukazuje percento kódu pokrytého testami. Existuje mnoho nástrojov (napr. JaCoCo, Cobertura), ktoré ti umožňujú analyzovať pokrytie kódu testami. Tieto nástroje ukážu, koľko percent tvojho kódu je pokrytého testami, čo ti umožní identifikovať časti kódu, ktoré nie sú testované.

**Odporúčame ti…**

*Pokrytie kódu je vždy časový kompromis medzi časom investovaným do vývoja programu a písania testov. Odporúčame preto snažiť sa pokryť 80% kódu unit testami.*

### Priebežne aktualizuj testy

Keď vykonávaš zmeny v kóde, nezabudni potom aktualizovať aj príslušné testy. Zachovávaj testovacie scenáre vždy aktuálne a uisti sa, že nové funkcie sú riadne otestované.

S týmito tipmi a trikmi by si mal byť pripravený začať písať efektívne Unit testy v jazyku Java. Zapamätaj si, že dobre napísané testy zlepšujú kvalitu kódu a dokážu ti ušetriť neskôr veľa času potrebného na analýzu chýb v kóde.