# Pokročilé programovací techniky

Kryštof Malinda, Ondřej Kohout, Stanislav Čech, Petr Svoboda

## Úkol 1

## Úkol 2

### Čast 1

V naší implementaci jsem vytvořil dvě statické metody comp pro třídu **ZpracovaniMy** podle zadání úkolu. Tyto metody slouží jako komparační funkce pro porovnání dvou hodnot - jednu pro porovnání celých čísel a druhou pro porovnání řetězců.

Pro metodu comp(int a, int b):

Implementoval jsem ji tak, aby vracela:

* -1 pokud je a menší než b
* 0 pokud je a rovno b
* 1 pokud je a větší než b

Použili jsem standardní podmínkové výrazy pro porovnání hodnot

Pro metodu comp(String a, String b):

Nejprve jsem ošetřil případy, kdy jeden nebo oba řetězce mohou být **null**

Pro samotné porovnání jsem využil vestavěnou metodu **compareTo**, která provádí porovnání řetězců

Tato metoda vrací:

* záporné číslo pokud je a lexikograficky menší než b
* 0 pokud jsou řetězce identické
* kladné číslo pokud je a lexikograficky větší než b

Tyto implementace jsou inspirovány použitím podobných metod jako **Double.compare()**

### Čast 2

Shrnutí odhalených chyb v implementacích:

1. V původní implementaci **VyhodnoceniProspechu**:

Metody **prospech** a **prumer** byly prázdné, neobsahovaly žádnou implementaci

V metodě **prumer** chyběla kontrola na **null** nebo **prázdné pole**

2. V původní implementaci **Zpracovani**:

* Metoda interval vracela vždy **false** místo správného vyhodnocení intervalu
* Metoda **vzorkySestupně** vracela vždy **null** místo generování pole hodnot
* Chyběly kontroly vstupních parametrů
* Chyběly dokumentační komentáře

3. V původní implementaci **ZpracovaniMoje**:

Obě metody comp vracely vždy -1 místo správného porovnání hodnot

Všechny tyto chyby byly opraveny v nových implementacích a byly přidány testy, které ověřují správnou funkčnost. Také byly přidány JavaDoc komentáře ke všem metodám a třídám.

## Úkol 3

## Úkol 4

## Úkol 5

### Čast 1

**Program implementuje následující třídy**

* KvR - základní třída reprezentující kvadratickou rovnici ve tvaru y = ax² + bx + c
  + Obsahuje koeficienty a, b, c
  + Poskytuje veřejné gettery pro všechny koeficienty
  + Poskytuje chráněné settery pro všechny koeficienty
* SpoctiYKvR - třída pro výpočet hodnot y kvadratické rovnice
  + Metoda calc(KvR, x) - vypočítá hodnotu y pro jednu hodnotu x
  + Metoda calcY(KvR, x[]) - vypočítá hodnoty y pro pole hodnot x
* KvREx - rozšířená třída KvR, která navíc obsahuje bod (x, y) ležící na křivce
  + Dědí vlastnosti třídy KvR
  + Přidává souřadnice x a y
  + Obsahuje metodu calcY() pro výpočet y souřadnice z aktuální hodnoty x
* SpoctiYKvRTest - testovací třída pro SpoctiYKvR
  + Testuje výpočet pro jednotlivé hodnoty
  + Testuje výpočet pro pole hodnot
  + Testuje speciální případy (prázdné pole, nulové koeficienty, atd.)

**Vysvětlení implementace**

* Proč chráněné (protected) settery v KvR?
  + Settery jsou označeny jako chráněné, aby je mohly využívat pouze odvozené třídy, ale ne externí kód
  + Tím je zajištěna zapouzdřenost a integritu třídy
* Vztah SpoctiYKvR a KvR
  + SpoctiYKvR využívá kompozice místo dědičnosti - dostane KvR jako parametr
  + Tento přístup umožňuje větší flexibilitu a lepší oddělení zodpovědností

Implementace KvREx

* KvREx rozšiřuje KvR a přidává souřadnice bodu na křivce
* V konstruktoru se automaticky vypočítá hodnota y pro zadané x
* Metoda calcY() počítá hodnotu y podle vzorce y = ax² + bx + c

**Testování**

* Třída SpoctiYKvRTest pokrývá základní funkčnost pro výpočet hodnot
* Testuje jednotlivé výpočty, pole hodnot i speciální případy

Kompletní pokrytí testy Pro kompletní pokrytí tříd KvR a KvREx testy by bylo třeba:

Pro KvR:

* Testovat konstruktor s různými hodnotami koeficientů
* Testovat všechny gettery
* Testovat všechny settery (to by vyžadovalo vytvoření testovací třídy, která dědí z KvR, aby měla přístup k protected metodám)
* Testovat hraniční hodnoty koeficientů

Pro KvREx:

* Testovat konstruktor s různými hodnotami koeficientů a souřadnice x
* Testovat všechny zděděné metody z KvR
* Testovat gettery pro souřadnice x a y
* Testovat metodu calcY() pro různé hodnoty koeficientů
* Ověřit, že hodnota y v konstruktoru je skutečně vypočítána podle vzorce

### Část 2

**Program je rozdělen do několika vrstev podle zadání**

**Datová vrstva**

* NacteniDat\_DataKvadratickeRovnice - abstraktní třída pro načítání dat
* NacteniDatCSV\_DataKvadratickeRovnice - konkrétní implementace načítání dat z CSV
* VystupData\_DataKvadratickeRovnice - abstraktní třída pro výstup dat
* VystupDataCSV\_DataKvadratickeRovnice - konkrétní implementace výstupu dat do CSV

**Logická vrstva:**

* KvR - třída reprezentující kvadratickou rovnici
* SpoctiYKvR - třída pro výpočet hodnot
* Provider - třída zpracovávající data, koordinuje načítání, výpočet a výstup

**Builder:**

* Builder - třída poskytující metodu pro vytvoření Provider objektu

**Hlavní třída:**

* Main - obsahuje metodu main, která spouští program

**Popis funkčnosti programu**

* Program načte jméno vstupního souboru z parametrů nebo od uživatele
* Vytvoří Provider pomocí Builder třídy
* Provider načte data ze vstupního souboru
* Pokud vstupní soubor neexistuje, požádá uživatele o zadání nového jména souboru

**Provede výpočty pro každý řádek dat:**

* Parsuje koeficienty a, b, c
* Vytvoří objekt KvR
* Pro každé x spočítá y pomocí SpoctiYKvR
* Sestaví výstupní řádek ve formátu a;b;c;x1;x2;...
* Zapíše výsledky do výstupního souboru

Ošetření chybových stavů Program ošetřuje následující chybové stavy:

Špatný formát dat ve vstupním souboru Neexistující vstupní soubor (program se zeptá na nový název) Chyby při zápisu do výstupního souboru

Formát vstupního souboru Vstupní soubor by měl být ve formátu CSV s oddělovačem ;: a1;b1;c1;x1 a2;b2;c2;x1;x2;x3 ... Kde:

a, b, c jsou koeficienty kvadratické rovnice y = ax² + bx + c x1, x2, ... jsou hodnoty x, pro které se mají vypočítat hodnoty y

Formát výstupního souboru Výstupní soubor bude ve formátu: a1;b1;c1;x1:y1 a2;b2;c2;x1:y1;x2:y2;x3:y3 ... Kde:

a, b, c jsou koeficienty kvadratické rovnice x jsou dvojice hodnot x a k nim vypočtených hodnot y

Program je navržen tak, aby byl snadno rozšiřitelný - můžete jednoduše přidat nové implementace načítání a výstupu dat (např. pro XML, JSON nebo databázi) bez nutnosti měnit logickou vrstvu.2 / 2RetryClaude does not have the ability to run the code it generates yet.Claude can make mistakes. Please double-check responses.

## Úkol 6

### Verze 1

Program implementuje třídu **Faktura** a další pomocné třídy, které spolu tvoří kompletní reprezentaci faktury. Využil jsem princip kompozice, kdy třída Faktura obsahuje reference na další objekty (Prijemce, CastkyDokladu, Sazba). Toto je správný objektově orientovaný přístup - každá třída má svou jasně definovanou zodpovědnost.

**Hlavní vlastnosti implementace**

* Nemutovatelnost objektů - Všechny třídy jsou implementovány jako immutable (neměnitelné). Proto mají všechny atributy modifikátor final a nejsou k dispozici žádné settery, což zajišťuje, že údaje na faktuře nelze měnit po vytvoření, jak bylo požadováno v zadání.
* Zapouzdření - Všechny atributy jsou privátní a přístup k nim je umožněn pouze přes gettery.
* Kompozice - Třída Faktura obsahuje objekty jiných tříd (Prijemce, CastkyDokladu), což je příklad kompozice objektů.
* Automatický výpočet - Třída CastkyDokladu automaticky vypočítá cenu s DPH na základě ceny bez DPH a sazby.
* Formátování - Metoda toString() je přepsána ve všech třídách pro pěkný a přehledný výpis.

**Testovací případy**

* V testovací třídě FakturaTest jsou implementovány tři základní testy:
  + Vytvoření standardní faktury - Testuje vytvoření a výpis faktury se základní sazbou DPH.
  + Faktura s nulovou hodnotou - Testuje, zda funguje korektně faktura s nulovou částkou.
  + Faktura se sníženou sazbou DPH - Testuje použití jiné sazby DPH.
  + Testy vypíšou výsledné faktury na standardní výstup, takže můžete zkontrolovat jejich správnost pohledem.

### Verze 2

**Vysvětlení úprav v implementaci**

Původní implementace již využívala kompozici objektů (třída Faktura obsahovala objekty tříd Prijemce a CastkyDokladu), ale nyní jsem přidal následující funkce k třídě Faktura:

* Zprostředkování dílčích hodnot - Do třídy Faktura jsem přidal metody, které delegují volání na metody kompozitních objektů. Například metoda getJmenoPrijemce() volá metodu getJmeno() na objektu prijemce. Tímto způsobem třída Faktura zprostředkovává přístup k dílčím hodnotám svých komponent.
* Aktualizace testů - V testovací třídě jsem přidal nový test testDilciHodnoty(), který demonstruje přímý přístup k dílčím hodnotám přes delegované metody třídy Faktura.
* UML diagram - Vytvořil jsem UML diagram, který znázorňuje vztahy mezi třídami v našem řešení. Kompozice je znázorněna plnou čárou s diamantem na straně celku.

**Změny v implementaci testů**

* S přidáním delegovaných metod do třídy Faktura se mění i způsob, jakým můžeme testy implementovat. Nyní můžeme testovat dílčí hodnoty přímo přes rozhraní třídy Faktura, aniž bychom museli pracovat s jejími vnitřními objekty.

**Hlavní rozdíly**

**Původní přístup (bez delegace):**

javaFaktura faktura = new Faktura(...);

String jmeno = faktura.getPrijemce().getJmeno();

double cenaBezDph = faktura.getCastkyDokladu().getCenaBezDph();

**Nový přístup (s delegací):**

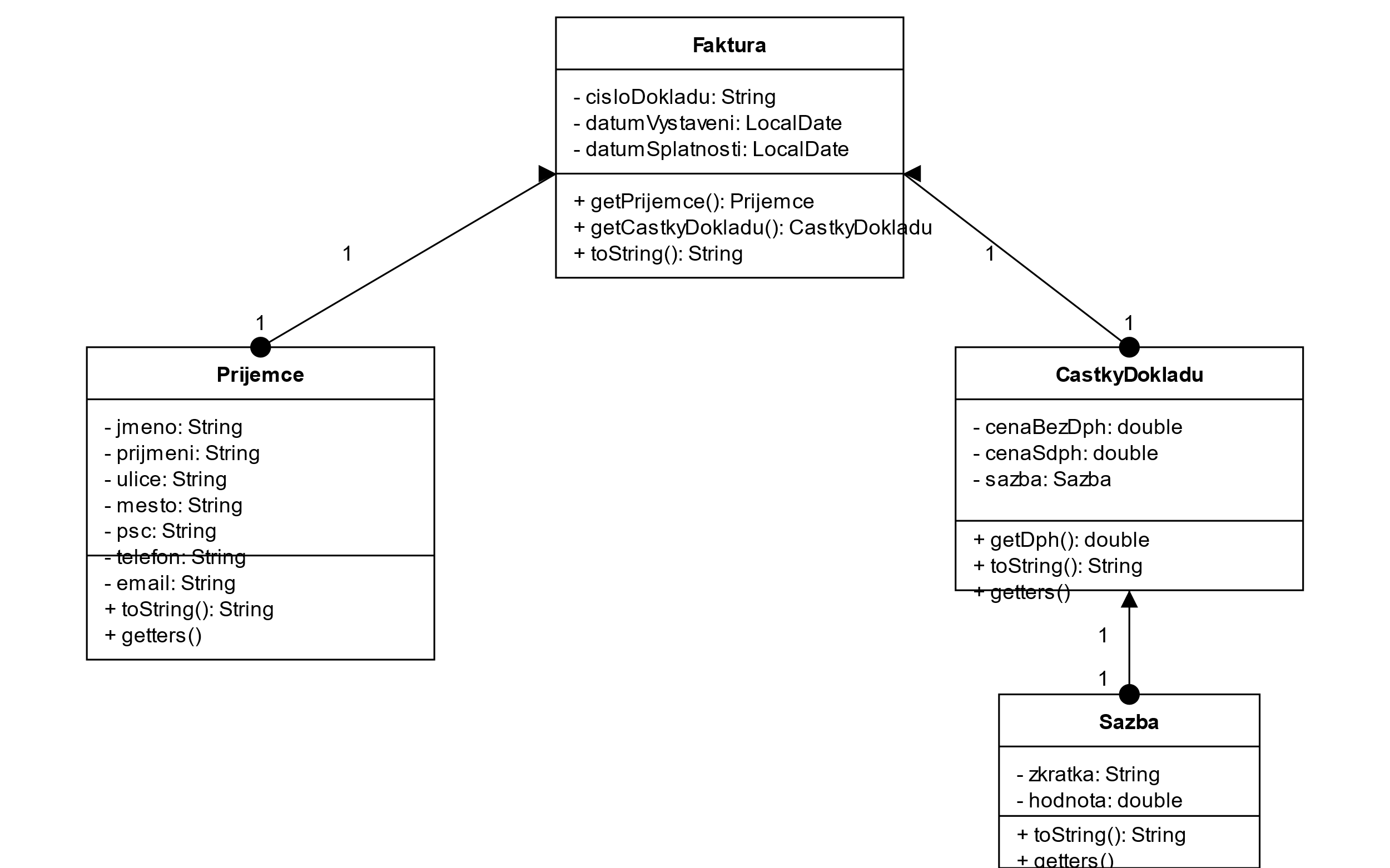
javaFaktura faktura = new Faktura(...);

String jmeno = faktura.getJmenoPrijemce();

double cenaBezDph = faktura.getCenaBezDph();

**Tento nový přístup má několik výhod:**

* Jednodušší API - Klient může přistupovat k dílčím hodnotám přímo přes rozhraní třídy Faktura, což je jednodušší a přímější.
* Abstrakce implementace - Klient nemusí znát vnitřní strukturu objektu Faktura.
* Flexibilita změn - Pokud se v budoucnu změní implementace vnitřních objektů, klientský kód nemusí být měněn.



## Úkol 7

## Úkol 8

## Úkol 9

## Úkol 10

### Fibonacciho posloupnosti

Tento projekt ukazuje **tři způsoby výpočtu n-tého prvku Fibonacciho posloupnosti**, indexované od 0:

1. calcNerek(int n) – **nerekurzivní přístup** (iterativní řešení).
2. calcRek(int n) – **rekurzivní přístup** (pomalejší, ukazuje princip).
3. calcRekTable(int n) – **dynamické programování** s použitím **tabulky (memoizace)**.

**Proč tři různé metody?**

**1. Iterativní (calcNerek)**

* Efektivní a rychlý způsob bez rekurze.
* Paměťově úsporný – používá jen dvě proměnné.

**2. Rekurzivní (calcRek)**

* Ukazuje princip, ale je **neefektivní pro větší n** (exponenciální časová složitost).
* Nepoužívá mezipaměť, dochází k opakovanému výpočtu stejných hodnot.

**3. Dynamické programování (calcRekTable)**

* **Efektivní kombinace rekurze a mezipaměti (tabulky)**.
* Tabulka ukládá mezivýsledky – **zabraňuje zbytečným výpočtům**.
* Výstup ukazuje stav tabulky v jednotlivých krocích, což usnadňuje sledování výpočtu.

**Co dělá init()?**

Metoda init(int n) inicializuje pole table, které slouží jako paměť pro již vypočítané hodnoty Fibonacciho čísel.

* Pole má velikost n + 1.
* Hodnoty jsou na začátku nastaveny na -1, což značí, že ještě nebyly vypočítány.

**Výhody použití tabulky**

* Zrychlení výpočtu – místo exponenciálního času (O(2^n)) se dostáváme na **lineární čas** O(n).
* Zabránění opakovaným výpočtům.
* Přehledný záznam průběhu výpočtu.

### Metody indexOf() v Java ArrayList

**Soubor IndexOfTest.java**

Obsahuje pět testovacích případů:

1. **První výskyt prvku**: Ověření, že metoda vrací index prvního výskytu zadaného prvku.
2. **Prvek neexistuje**: Ověření, že metoda vrací -1, pokud prvek není v seznamu.
3. **Prázdný seznam**: Ověření, že metoda vrací -1 pro prázdný seznam.
4. **Prvek je null**: Ověření, že metoda správně identifikuje index prvku null.
5. **Více výskytů prvku**: Ověření, že metoda vrací index prvního výskytu i při více výskytech.

### Porovnání řetězců v Javě

Implementovat vlastní metodu pro lexikografické porovnání dvou řetězců bez použití vestavěných metod jako equals() nebo compareTo().

**Soubor StringUtils.java**

Obsahuje metodu customCompare, která:

* Porovnává dva řetězce znak po znaku.
* Vrací:
  + 0 pokud jsou řetězce stejné.
  + -1 pokud je první řetězec lexikograficky menší.
  + 1 pokud je první řetězec lexikograficky větší.

**Testování**

Soubor StringUtilsTest.java obsahuje metodu main, která testuje customCompare na různých dvojicích řetězců, včetně:

* "ABCDEF" vs "ABCD" → očekáváno 1
* "ABCD" vs "ABCDEF" → očekáváno -1
* "ABCD" vs "ABCD" → očekáváno 0
* "ABCD" vs "ABCE" → očekáváno -1
* "ABCE" vs "ABCD" → očekáváno 1