02. Znaky, řetězce a textové soubory

Obsah

- Reprezentace znaků a řetězců v paměti
- Kódování znaků (ASCII, Unicode, UTF-8)
- Základní operace s řetězci
- Porovnávání řetězců
- Vyhledávání podřetězce
- Práce s textovými soubory
- Výjimky
- Efektivita, příklady

Reprezentace znaků a řetězců v paměti

Znaky v Javě

V Javě jsou znaky reprezentovány datovým typem `char`, který je 16bitový a může obsahovat libovolný znak z Unicode do hodnoty 65 535.

```
char znak = 'A'; // reprezentováno číslem 65
int hodnota = (int) znak; // převod na číselnou hodnotu
```

Řetězce v Javě

Řetězce jsou v Javě reprezentovány třídou `String`. Tato třída je immutable (neměnná), což znamená, že jednou vytvořený řetězec se nemůže změnit.

Pro změny řetězců se používají třídy `StringBuilder` (není synchronizovaná) a `StringBuffer` (je synchronizovaná).

```
String text = "Hello";
text = text + " World"; // vytvoří se nový objekt, původní zůstává nezměněn

// Efektivnější způsob pro mnoho operací
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" World");
String result = sb.toString();
```

Kódování znaků

ASCII (American Standard Code for Information Interchange)

- 7bitové kódování (hodnoty 0-127)
- Obsahuje anglickou abecedu, číslice, interpunkci a řídící znaky
- Nevýhoda: Podporuje pouze znaky anglické abecedy
- Rozšířená ASCII (8 bitů) umožňuje kódovat 256 znaků

Unicode

- Univerzální znakový standard
- Podporuje přes 140 000 znaků z mnoha písem a jazyků (všechny existující písma)
- Každý znak má jednoznačný kód (code point)
- Základní rovina (BMP - Basic Multilingual Plane) obsahuje znaky s kódy 0-65535
- Existují další roviny s vyššími kódy (např. emoji, historická písma)

UTF-8 (Unicode Transformation Format)

- Proměnná délka kódování (1 až 4 bajty na znak)

- Efektivní pro latinku (1 bajt pro ASCII znaky)
- Kompatibilní s ASCII
- Nejčastěji používané kódování na webu
- Většina znaků latinky zabírá 1 bajt, další evropské znaky 2 bajty, asijské znaky 3 bajty

UTF-16

- Používáno v Javě pro interní reprezentaci řetězců
- Používá 2 nebo 4 bajty na znak
- Základní rovina Unicode (znaky pod U+10000) zabírá 2 bajty
- Znaky nad U+10000 jsou kódovány pomocí surrogate pairs (4 bajty)

UTF-32

- Používá vždy 4 bajty na znak
- Nevýhoda: Neefektivní využití paměti, zejména pro latinku
- Výhoda: Snadný přístup k jednotlivým znakům (index * 4)

Základní operace s řetězci

```
// Vytvoření řetězce
String s1 = "Hello";
String s2 = new String("World");

// Spojování řetězců
String s3 = s1 + " " + s2;  // "Hello World"
String s4 = s1.concat(" ").concat(s2);  // "Hello World"

// Délka řetězce
int délka = s1.length();  // 5

// Získání znaku na pozici
char znak = s1.charAt(2);  // 'l'

// Podřetězec
String podřetězec = s1.substring(1, 4);  // "ell" (indexy 1 až 3)

// Rozdělení řetězce
String text = "jablko,hruška,banán";
String[] ovoce = text.split(",");  // ["jablko", "hruška", "banán"]

// Převod na velká/malá písmena
String velká = s1.toUpperCase();  // "HELLO"
String malá = s1.toLowerCase();  // "hello"

// Odstranění mezer na začátku a konci
String s5 = "  text s mezerami  ";
String ořezaný = s5.trim();  // "text s mezerami"

// Nahrazení znaků nebo podřetězců
String nahrazený = s1.replace('l', 'x');  // "Hexxo"
String nahrazenýSubstring = s1.replace("el", "AL");  // "HALlo"
```

Porovnávání řetězců

```
String s1 = "Hello";
String s2 = "hello";
```

```java
// Rovnost (s rozlišením velikosti písmen)
boolean jsou_stejné = s1.equals(s2);  // false

// Rovnost (bez rozlišení velikosti písmen)
boolean jsou_stejné_case_insensitive = s1.equalsIgnoreCase(s2);  // true

// Porovnání podle abecedy (lexikograficky)
int výsledek = s1.compareTo(s2);  // záporné, protože 'H' < 'h'
// výsledek < 0 znamená s1 < s2
// výsledek == 0 znamená s1 == s2
// výsledek > 0 znamená s1 > s2

// Začátek a konec řetězce
boolean začíná = s1.startsWith("He");  // true
boolean končí = s1.endsWith("lo");  // true

// Obsahuje podřetězec
boolean obsahuje = s1.contains("ell");  // true
```

Vyhledávání podřetězce

Základní metody

```java
String text = "Ahoj, jak se máš? Ahoj.";

// Najít první výskyt
int pozice = text.indexOf("Ahoj");  // 0
int druhýVýskyt = text.indexOf("Ahoj", 1);  // 17

// Najít poslední výskyt
int poslednípozice = text.lastIndexOf("Ahoj");  // 17
```

Algoritmy vyhledávání podřetězce

Naivní algoritmus Časová složitost: O(n*m), kde n je délka textu a m je délka hledaného podřetězce.

```java
// Naivní algoritmus pro vyhledávání podřetězce
int vyhledejPodřetězec(String text, String vzor) {
    int n = text.length();
    int m = vzor.length();

    for (int i = 0; i <= n - m; i++) {
        int j;
        for (j = 0; j < m; j++) {
            if (text.charAt(i + j) != vzor.charAt(j)) {
                break;
            }
        }

        if (j == m) {
            return i;  // nalezen vzor na pozici i
        }
    }

    return -1;  // vzor nebyl nalezen
}
```

Knuth-Morris-Pratt algoritmus Efektivnější algoritmus s časovou složitostí O(n+m).

```
// Knuth-Morris-Pratt algoritmus
int kmpVyhledávání(String text, String vzor) {
    int n = text.length();
    int m = vzor.length();

    // Příprava pomocné tabulky
    int[] lps = new int[m];
    int j = 0;

    // Výpočet LPS (Longest Proper Prefix which is also Suffix) pole
    výpočetLpsArray(vzor, m, lps);

    int i = 0;  // index pro text
    while (i < n) {
        if (vzor.charAt(j) == text.charAt(i)) {
            j++;
            i++;
        }

        if (j == m) {
            return i - j;  // vzor nalezen na pozici i-j
        } else if (i < n && vzor.charAt(j) != text.charAt(i)) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }

    return -1;  // vzor nebyl nalezen
}

// Pomocná metoda pro KMP
void výpočetLpsArray(String vzor, int m, int[] lps) {
    int len = 0;
    int i = 1;
    lps[0] = 0;

    while (i < m) {
        if (vzor.charAt(i) == vzor.charAt(len)) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

Práce s textovými soubory

Čtení ze souboru

```
try {
    // Základní čtení souboru
    String obsah = Files.readString(Path.of("soubor.txt"));

    // Čtení po řádcích
    List<String> řádky = Files.readAllLines(Path.of("soubor.txt"));

    // Stream API
    Files.lines(Path.of("soubor.txt")).forEach(řádek -> {
        // zpracování řádku
    });

    // BufferedReader (pro větší soubory)
    try (BufferedReader br = new BufferedReader(new FileReader("soubor.txt"))) {
        String řádek;
        while ((řádek = br.readLine()) != null) {
            // zpracování řádku
        }
    }
} catch (IOException e) {
    // zpracování výjimky
}
```

Zápis do souboru

```
try {
    // Základní zápis
    Files.writeString(Path.of("vystup.txt"), "Hello world");

    // Zápis více řádků
    List<String> řádky = Arrays.asList("Řádek 1", "Řádek 2", "Řádek 3");
    Files.write(Path.of("vystup.txt"), řádky);

    // BufferedWriter (pro větší soubory)
    try (BufferedWriter bw = new BufferedWriter(new FileWriter("vystup.txt"))) {
        bw.write("První řádek");
        bw.newLine();
        bw.write("Druhý řádek");
    }
} catch (IOException e) {
    // zpracování výjimky
}
```

Výjimky při práci s řetězci a soubory

Běžné výjimky při práci s řetězci

- `StringIndexOutOfBoundsException` - přístup k indexu mimo rozsah řetězce
- `NullPointerException` - pokus o operaci s null řetězcem

Výjimky při práci se soubory

- `IOException` - obecná výjimka pro I/O operace
- `FileNotFoundException` - soubor nebyl nalezen
- `SecurityException` - nedostatečná oprávnění

- `UnsupportedEncodingException` - nepodporované kódování

Příklad ošetření výjimek

```
try {
    String obsah = Files.readString(Path.of("soubor.txt"));
    // zpracování obsahu
} catch (FileNotFoundException e) {
    System.err.println("Soubor nebyl nalezen: " + e.getMessage());
} catch (IOException e) {
    System.err.println("Chyba při čtení souboru: " + e.getMessage());
} finally {
    // kód, který se provede vždy
}

// Nebo s try-with-resources
try (BufferedReader br = new BufferedReader(new FileReader("soubor.txt"))) {
    String radek;
    while ((radek = br.readLine()) != null) {
        // zpracování řádku
    }
} catch (IOException e) {
    // zpracování výjimky
}
```

Praktické příklady

Počítání frekvence znaků v textu

```
Map<Character, Integer> počítejFrekvenci(String text) {
    Map<Character, Integer> frekvence = new HashMap<>();

    for (char c : text.toCharArray()) {
        frekvence.put(c, frekvence.getOrDefault(c, 0) + 1);
    }

    return frekvence;
}
```

Nalezení nejdelšího slova v souboru

```
String nejdelšíSlovo(String cestaKSouboru) throws IOException {
    String nejdelší = "";

    try (BufferedReader br = new BufferedReader(new FileReader(cestaKSouboru))) {
        String řádek;
        while ((řádek = br.readLine()) != null) {
            String[] slova = řádek.split("\\s+");

            for (String slovo : slova) {
                // Očistíme slovo od interpunkce
                slovo = slovo.replaceAll("[^a-zA-Z0-9]", "");

                if (slovo.length() > nejdelší.length()) {
                    nejdelší = slovo;
                }
            }
        }
```

```
    }

    return nejdelší;
}
```

Počítání výskytů podřetězce v textu

```
int počítejVýskyty(String text, String vzor) {
    int počet = 0;
    int pozice = 0;

    while ((pozice = text.indexOf(vzor, pozice)) != -1) {
        počet++;
        pozice += vzor.length();
    }

    return počet;
}
```

Efektivita operací s řetězci

Přidávání ke stringu v cyklu

```
// Neefektivní způsob
String výsledek = "";
for (int i = 0; i < 10000; i++) {
    výsledek += i;  // vytváří nový objekt při každé iteraci
}

// Efektivní způsob
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10000; i++) {
    sb.append(i);
}
String výsledek = sb.toString();  // pouze jeden nový objekt String
```

Porovnání efektivity vyhledávacích algoritmů

- Naivní algoritmus: O(n*m) - jednoduchý, ale neefektivní pro velké texty
- KMP algoritmus: O(n+m) - efektivnější, ale složitější implementace
- Boyer-Moore algoritmus: O(n/m) v nejlepším případě - velmi efektivní pro dlouhé vzory

String pooling v Javě

Java uchovává literální řetězce v tzv. "string pool", kde jsou sdíleny mezi různými částmi programu:

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = new String("Hello");

boolean b1 = (s1 == s2);  // true - oba odkazují na stejný objekt v poolu
boolean b2 = (s1 == s3);  // false - s3 je nový objekt mimo pool
boolean b3 = s1.equals(s3);  // true - obsahově stejné
```