Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

KIERUNEK STUDIÓW INFORMATYKA

MATERIAŁY DO ZAJĘĆ LABORATORYJNYCH

Programowanie w języku SWIFT

Autor:
dr inż. Maria Skublewska-Paszkowska

Lublin 2021

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

LABORATORIUM 3. PROGRAMY DZIAŁAJĄCE NA NAPISACH I LITERAŁACH ZNAKOWYCH.

Cel laboratorium:

Poznanie łańcuchów w języku Swift.

Zakres tematyczny zajęć:

- typ *String*,
- inicjalizacja łańcucha,
- literały znakowe,
- operacja na łańcuchach,
- dostęp i modyfikacja łańcuchów.
- porównywanie łańcuchów.

Pytania kontrolne:

1. Co typ String może zawierać?
2. W jaki sposób można przechowywać łańcuchy wieloliniowe?
3. Jakie znaki specjalne można zastosować wewnątrz łańcucha?
4. W jaki sposób można zainicjować pusty ciąg?
5. Jaka jest różnica między łańcuchem, a tablicą znakową?
6. W jaki sposób można utworzyć nowy łańcuch?
7. Na czym polega interpolacja łańcuchów?
8. Jakie są podstawowe metody manipulacji łańcuchami?
9. W jaki sposób można porównać łańcuchy?

Łańcuchy w języku Swift są reprezentowane poprzez typ **String**. Zawartość tego typu może być reprezentowana także jako znak, czy kolekcja znaków. Składnia jest prosta i podobna do języka C. Każdy ciąg składa się ze znaków Unicode niezależnych od kodowania i zapewnia obsługę dostępu do tych znaków w różnych reprezentacjach Unicode. Ciąg jest reprezentowany jako łańcuch znakowy ujęty w cudzysłowy. Ciąg przechowywany jako zmienna jest modyfikowalny (ang. *mutable*), natomiast w postaci stałej – nie.

Zadeklarowany ciąg może być ciągłym tekstem lub podzielony na linie (ang. *multiline*) (Listing 3.1). Jeśli ciąg ma być wieloliniowy, należy zastosować potrójny cudzysłów. Wieloliniowe łańcuch zawierają także puste linie, co oznacza, że podział linii jest zwartością łańcucha. Można także wstawić znak *\n*, aby wymusić przejście do nowej linii (Listing 3.2).

Listing 3.1. Przykłady ciągów prostych i wieloliniowych

```
let str = "Napis jednoliniowy"
let mstr = """
    Pierwsza linia

    Druga linia
    Trzecia linia
    """
```

Listing 3.2. Ciąg z przejściem do nowych linii

```
let mstr = """
     Pierwsza linia\n Druga linia \n Trzecia linia
    """
```

Ciąg wielowierszowy można wciąć. Biała spacja przed zamykającymi cudzysłowami oznacza, jakie białe znaki ma zignorować przed wszystkimi innymi wierszami. Jeśli jednak białe znaki będą na początku wiersza oprócz tego, co znajduje się przed zamykającymi cudzysłowami, ta biała spacja będzie elementem łańcucha.

Wewnątrz łańcucha można stosować specjalne znaki, takie jak:

- \0 – pusty znak;
- \\ – odwrotny ukośnik,
- \t – tabulacja;
- \n – wcięcie wiersza;
- \r – enter;
- \" – podwójny cudzysłów;
- \' – pojedynczy cudzysłów;
- wartość skalarna Unicode, zapisana jako *\u{n}*, gdzie n to 1–8 cyfrowa liczba szesnastkowa.

Na Listingu 3.3. przedstawiono przykłady łańcuchów z zastosowaniem znaków specjalnych.

Listing 3.3. Ciąg ze znakami specjalnymi

```
let cytat = " \"Lepiej zaliczać się do niektórych, niż do
              wszystkich \" Andrzej Sapkowski "
let tab = "\t Linia z tabluatorem"
let wciecie = "\tn Linia z wcięciem"
let znak1 = "\u{0040}"  // symbol at
let znak2 = "\u{1F31F}" //gwiazda
```

Inicjalizację pustego ciągu można uzyskać stosując podwójne cudzysłowy lub poprzez inicjalizację typu *String* (Listing 3.4). Do sprawdzenia, czy ciąg jest pusty, można użyć funkcji *isEmpty* (Listing 3.4).

Listing 3.4. Inicjalizacja pustego ciągu

```
//inicjalizacja pustych ciągów
var str1 = ""
var str2 = String()

print(str1, str2)

// sprawdzenie, czy  ciąg jest pusty
if str1.isEmpty {
    print("Ciąg jest pusty!")
}
```

Jeśli utworzony ciąg jest przypisywany do stałej lub zmiennej, czy jest przekazywany do funkcji, jego wartość jest kopiowana. Nowa kopia wartości jest przekazywana, podczas gdy oryginalna wartość nie jest zmieniana. Oznacza to, że oryginalna wartość nie zostanie zmodyfikowana, chyba, że programista sam tego dokona.

Dostęp do poszczególnych **literałów znakowych** odbywa się za pomocą pętli (Listing 3.5). Każdy znak zostanie pobrany i wyświetlony w kolejnych liniach.

Listing 3.5. Dostęp do literałów znakowych

```
let str = "Mały miś"
for char in str {
    print(char)
}
```

Literał znakowy jest deklarowany jako typ *Character* (Listing 3.6).

Listing 3.6. Definicja literału znakowego

```
let char = „a"
```

Ciąg można zadeklarować jako tablica literałów (Listing 3.7). Na jej bazie można utworzyć nowy tekst typu *String*. Wyświetlenie obu ciągów daje różną ich reprezentację.

Listing 3.7. Ciąg literałów i typ tekstowy

```
let chars: [Character] = ["s", "t", "u", "d", "e", "n", "t"]
let str = String(chars)
print(chars) // ["s", "t", "u", "d", "e", "n", "t”]
print(str) // student
```

Konkatenacja łańcuchów i literałów znakowych można przeprowadzić za pomocą znaku +, += oraz funkcji *append()*. Operator + tworzy nowy ciąg, natomiast dwa pozostałe pozwalają na dodanie nowego ciągu na końcu istniejącego. Należy pamiętać o umieszczeniu spacji. Poszczególne operatory zostały przedstawione na Listingach 3.8, 3.9 oraz 3.10.

Listing 3.8. Konkatenacja ciągów z zastosowaniem operatora +

```
let w = "Witaj "
let n = "Janek"
var wn = w + n
print(wn) // Witaj Janek
```

Listing 3.9. Konkatenacja ciągów z zastosowaniem operatora +=

```
var w = "Witaj "
let n = "Janek"
w += n
print(w) // Witaj Janek
```

Listing 3.10. Konkatenacja ciągów z zastosowaniem funkcji append()

```
var w = "Witaj "
let n = "Janek"
w += n
let z = "!"
w.append(z)
print(w) // Witaj Janek!
```

Interpolacja łańcuchów pozwala utworzyć nowy ciąg na podstawie różnych stałych, zmiennych, literałów i wyrażeń poprzez uwzględnienie ich wartości wewnątrz literału ciągu. Dotyczy ona zarówno ciągów jedno- i wieloliniowych. Na Listingu 3.11 przedstawiono tworzenie dwóch ciągów. W pierwszym bok kwadratu jest wyświetlany jako jego element, ale także posłużył do obliczenia wyświetlonej wartości. Należy upewnić się, że zmienna jest określonego typu. Drugi ciąg przedstawia obliczenie wartości na podstawie zmiennych.

Listing 3.11. Interpolacja łańcuchów

```
let bok = 12
let str1 = "Obwód kwadratu o boku \(bok)
            wynosi \(Int(bok)*4) "
let str2 = "Pole kwadratu o boku \(bok)
            wynosi \((bok)*(bok)) "
print(str1) // Obwód kwadratu o boku 12 wynosi 48
print(str2) // Pole kwadratu o boku 12 wynosi 144
```

Rozszerzona interpolacja łańcuchów umożliwia wyświetlanie zmiennych, a nie ich wartości, co zostało przedstawione na Listingu 3.12.

Listing 3.12. Rozszerzona interpolacja łańcuchów

```
print(#"Obwód kwadratu o boku \(bok)wynosi \(Int(bok)*4) "#)
// Obwód kwadratu o boku \(bok)wynosi \(Int(bok)*4)
```

Zliczanie elementów łańcucha wykonywane jest za pomocą funkcji *count*. Na listingu 3.13 przedstawiono zliczenie znaków zdefiniowanego ciągu.

Listing 3.13. Zliczenie elementów łańcucha

```
let str = "Nowy ciąg"
print("Liczba zanków w ciągu \(str) wynosi \(str.count)”)
// Liczba zanków w ciągu Nowy ciąg wynosi 9
```

Za pomocą typu **String.index** można uzyskać dostęp do poszczególnych elementów znakowych. Ponieważ znaki mogą być różnego rozmiaru, do przechodzenia po poszczególnych literałach należy użyć skalary Unicode od początku (*start*) do końca ciągu (*end*). Nie można zastosować całkowitych indeksów łańcucha. W przypadku, gdy łańcuch jest pusty, *startIndex* oraz *endIndex* wskazują tą samą wartość.

Dostęp do elementów znakowych:

- właściwość *startIndex* – zwraca pozycję pierwszego znaku;
- właściwość *endIndex* – zwraca pozycję po ostatnim znaku i dlatego jego wywołanie spowoduje błąd;
- metoda *index(after:)* – zwraca pozycję znaku po podanym;
- metoda *index(before:)* – zwraca pozycję znaku przed podanym;
- metoda *index(_:offsetBy:)* – zwraca pozycję znaku po podanym i oddalonym o podany offset.

Dostęp do elementów został przedstawiony na Listingu 3.14.

Listing 3.14. Dostęp do elementów łańcucha

```
let str = "Programowanie w jezyku Swift"
print(str[str.startIndex]) // P
print(str[str.endIndex]) // Błąd
print(str[str.index(after: str.startIndex)]) // r
print(str[str.index(before: str.endIndex)]) // t
print(str[str.index(str.startIndex, offsetBy: 3)]) // g
```

Wstawienie pojedynczego znaku do ciągu na zadanym indeksie odbywa się za pomocą metody *insert(_:at:)*, a innego ciągu – metodą *insert(contentsOf:at:)*. Przykładowe manipulacje ciągu zostały przedstawione na Listingu 3.15.

Listing 3.15. Wstawianie znaku i ciągu

```
var str = "Studencie"
str.insert("!", at: str.endIndex)
print(str) //Studencie!
str.insert(contentsOf: " Witaj na zajęciach z programowania
Swift", at: str.endIndex)
print(str)
//Studencie! Witaj na zajęciach z programowania Swift
str.insert(contentsOf: " w języku", at:
                        str.index(str.endIndex, offsetBy: -6))
print(str)
//Studencie! Witaj na zajęciach z programowania w języku Swift
```

Usuwanie pojedynczego znaku z ciągu na zadanym indeksie odbywa się za pomocą metody *remove(at:)*, a łańcucha – metodą *removeSubrange(_:)*, co przedstawiono na Listingu 3.16, jako kontynuacja wcześniejszego kodu. W celu usunięcia ciągu należy zdefiniować zakres (range) podając indeks, od którego zostanie usunięty ciąg oraz indeks, na którym należy zakończyć.

Listing 3.16. Usuwanie znaku i ciągu

```
str.remove(at: str.index(str.startIndex, offsetBy: 9))
print(str)
//Studencie Witaj na zajęciach z programowania w języku Swift
let range = str.index(str.endIndex, offsetBy: -15) ..<
                        str.index(str.endIndex, offsetBy: -6)
str.removeSubrange(range)
print(str)
//Studencie Witaj na zajęciach z programowania Swift
```

Porównanie ciągów wykonywane jest na trzy sposoby:

- równości ciągów i znaków;
- równości prefiksów;
- równości sufiksów.

Sprawdzenie, czy dwa ciągi są identyczne, przedstawiono na Listingu 3.17. Porównanie zachodzi przez instrukcję *if*. Porównanie znaków w Unicode zostało przedstawione na Listingu 3.18. W pierwszym przypadku ciągi są identyczne, a w drugim program zwraca informację, że znaki są różne.

Listing 3.17. Porównanie ciągów

```
let str1 = "Programowanie - Swift"
let str2 = "Programowanie - Swift"
if str1 == str2 {
    print("Oba ciągi są identyczne")
}
else{
    print("Ciągi są różne")
}
```

Listing 3.18. Porównanie znaków

```
let latinChar: Character = "\u{41}" //A
let cyrillicChar: Character = "\u{0410}" //A
if latinChar == cyrillicChar {
    print("Oba ciągi są identyczne")
}
else{
    print("Ciągi są różne")
}
```

W celu sprawdzenia, czy ciąg ma określony prefiks lub sufiks ciągu, stosowane są metody *hasPrefix(_:)* i *hasSuffix(_:)*. Każda z tych metod przyjmuje ciąg jako parametr. Zwraca wartość typu logicznego. Przykład zastosowania tych metod przedstawiono na Listingu 3.19.

Listing 3.19. Porównanie prefiksów i sufiksów

```
let str1 = "Lab. Programowanie w języku Swift"
let str2 = "Lab"
let str3 = "ft"
if str1.hasPrefix(str2) {
    print("Ciąg: \(str1) zawiera prefiks \(str2)" )
}
if str1.hasSuffix(str3) {
    print("Ciąg: \(str1) zawiera sufiks \(str3)" )
}
```

Zadanie 3.1. Utwórz aplikację konsolową – napisy jedno i wieloliniowe

Polecenie 1. Utwórz program, który zdefiniuje dwa łańcuchy: jednoliniowy i wieloliniowy, a następnie je wyświetli.

Polecenie 2. Sprawdź działanie programu.

Zadanie 3.2. Utwórz aplikację konsolową – formatowanie tekstu

Polecenie 1. Utwórz program, który wyświetli sformatowany tekst w postaci:

> ”Nauka kodowania to nie tylko nauka języka technologii.
>
> To odkrywanie nowych sposobów myślenia
>
> i urzeczywistnianie rozmaitych koncepcji.”

Przed każdą linią jest wstawiona pusta linia. Analogicznie po ostatnim wierszu. Każdy wiersz zaczyna się tabulatorem.

Polecenie 2. Sprawdź działanie programu.

Zadanie 3.3. Utwórz aplikację konsolową – łączenie tekstów

Polecenie 1. Utwórz program, który wczyta od użytkownika: imię, drugie imię, nazwisko i rok urodzenia. Należy utworzyć nowy ciąg i go wyświetlić.

Polecenie 2. Usuń drugie imię. Wyświetl otrzymany ciąg.

Polecenie 3. Usuń rok urodzenia. Dodaj wiek osoby. Wyświetl otrzymany ciąg.

Polecenie 4. Sprawdź, czy imię osoby rozpoczyna się literą *D*.

Polecenie 5. Sprawdź działanie programu.

Zadanie 3.4. Utwórz aplikację konsolową – szukanie znaku

Polecenie 1. Utwórz program, który wczyta od użytkownika dowolny ciąg, pojedynczy znak oraz liczbę całkowitą. Program powinien sprawdzić, czy wczytany znak znajduje się na początku ciągu, na końcu, czy na indeksie oddalonym o podaną liczbę całkowitą od początku lub od końca ciągu.

Polecenie 2. Sprawdź działanie programu.

Zadanie 3.5. Utwórz aplikację konsolową – porównywanie ciągów

Polecenie 1. Utwórz program, który wczyta od użytkownika dwa ciągi i sprawdzi, czy oba ciągi są identyczne.
Polecenie 2. Wczytaj prefiks od użytkownika. Należy sprawdzić, czy prefiks występuje w podanych ciągach.
Polecenie 3. Wczytaj sufiks od użytkownika. Należy sprawdzić, czy prefiks występuje w podanych ciągach.

Polecenie 4. Sprawdź działanie programu.

Zadanie 3.6. Utwórz aplikację konsolową – kalkulator walut

Polecenie 1. Utwórz program, który wczyta od użytkownika kwotę w PLN, a następnie obliczy jego wartość w dolarach. Można przyjąć przelicznik: 1 USD = 3.9 PLN. W wyniku proszę uwzględnić symbol dolara.

Polecenie 2. Sprawdź działanie programu.

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego