Rust

Closure, iteratori

Closure

- Anonimne funkcije koje možete da dodelite promenljivoj ili da ih prosledite kao argument drugim funkcijama
- Možete da ih kreirate na jednom mestu a zatim da ih pozovete na drugom mestu kako biste ga izvršili u drugom kontekstu
- Za razliku od funkcija mogu da uzmu vrednost iz opsega u kome su definisane.

Closure - čuvanje vrednosti iz opsega

```
#[derive(Debug, PartialEq, Copy, Clone)]
enum ShirtColor {
    Red,
    Blue,
}

struct Inventory {
    shirts: Vec<ShirtColor>,
}
```

Closure - čuvanje vrednosti iz opsega

```
impl Inventory {
    fn giveaway(&self, user_preference: Option<ShirtColor>) -> ShirtColor {
        user_preference.unwrap_or_else(|| self.most_stocked())
    }
    fn most_stocked(&self) -> ShirtColor {
        let mut num_red = 0;
        let mut num_blue = 0;

        for color in &self.shirts {
            match color {
                ShirtColor::Red => num_red += 1,
                ShirtColor::Blue => num_blue += 1,
            }
        }
        if num_red > num_blue {
            ShirtColor::Red
        } else {
            ShirtColor::Blue
        }
    }
}
```

Closure - čuvanje vrednosti iz opsega

```
fn main() {
    let store = Inventory {
        shirts: vec![ShirtColor::Blue, ShirtColor::Red, ShirtColor::Blue],
    };

    let user_pref1 = Some(ShirtColor::Red);
    let giveaway1 = store.giveaway(user_pref1);
    println!("The user with preference {:?} gets {:?}",user_pref1, giveaway1);

    let user_pref2 = None;
    let giveaway2 = store.giveaway(user_pref2);
    println!("The user with preference {:?} gets {:?}",user_pref2, giveaway2);
}
```

Closure

- Ne zahteva da označite tipove parametara ili povratnu vrednost kao što to rade funkcije.

```
let expensive_closure = |num: u32| -> u32 {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
};
```

Closure

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

Closure

```
let example_closure = |x| x;


let s = example_closure(String::from("hello"));

let n = example_closure(5);
```

Ovaj kod se neće kompajlirati zato što kada prvi put pozovemo *closure* kompajler će zaključiti da je tip parametra *x, String,* a posle pozivamo *closure* sa drugim tipom.

Closure - referenca i prenos vlasništva

- *Closure* može da uzme vrednost iz okruženja na 3 načina:
  - Nepromenljivo pozajmljivanje
  - Promenljivo pozajmljivanje
  - Preuzimanje vlasništva
- Na koji način preuzima vrednost odlučiće na osnovu onoga što telo *closure-a* radi sa uzetom vrednošću.

Closure - nepromenljivo pozajmljivanje

```
fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let only_borrows = || println!("From closure: {:?}", list);

    println!("Before calling closure: {:?}", list);
    only_borrows();
    println!("After calling closure: {:?}", list);
}
```

Closure - promenljivo pozajmljivanje

```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let mut borrows_mutably = || list.push(7);

    borrows_mutably();
    println!("After calling closure: {:?}", list);
}
```

Closure - preuzimanje vlasništva

- Ako želite da prisilite *closure* da preuzme vlasništvo iako telu *closure-a* nije striktno potrebno vlasništvo, možete da upotrebite ključnu reč ***move*** pre liste parametara.
- Koristi se prilikom rada sa nitima

Closure - preuzimanje vlasništva

```
use std::thread;

fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    thread::spawn(move || println!("From thread: {:?}", list))
        .join()
        .unwrap();
}
```

Closure - rukovanje vrednostima iz okruženja

- Zavisi od *Fn* osobina koje *closure* implementira.
  - *FnOnce* - primenjuje se na *closure-ima* koji se mogu pozvati samo jednom.
    - Svi implementiraju bar ovu osobinu, jer se svi *closure-i* mogu pozvati
    - *Closure* koji pomera vrednost iz okruženja
  - *FnMut* - primenjuje se na *closure-ima* koji ne pomeraju snimljene vrednosti iz svog tela, ali može da menja vrednost iz okruženja.
    - Mogu da se pozovu više puta
  - *Fn* - primenjuje se na *closure-ima* koji ne pomeraju i ne menjaju vrednosti iz okruženja, kao i na *closure* koji ne uzimaju ništa iz svog okruženja.
    - Mogu biti pozvani više puta bez mutiranja njihovog okruženja

Closure - rukovanje vrednostima iz okruženja

```
impl<T> Option<T> {
    pub fn unwrap_or_else<F>(self, f: F) -> T
    where
        F: FnOnce() -> T
    {
        match self {
            Some(x) => x,
            None => f(),
        }
    }
}
```

Closure - rukovanje vrednostima iz okruženja - *FnMut*

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    list.sort_by_key(|r| r.width);
    println!("{:#?}", list);
}
```

Closure - rukovanje vrednostima iz okruženja - *FnOnce*

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];
    let mut sort_operations = vec![];
    let value = String::from("by key called");
    list.sort_by_key(|r| {
        sort_operations.push(value);
        r.width
    });
    println!("{:#?}", list);
}
```

```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];
    let mut sort_operations = vec![];
    let value = String::from("by key called");
    list.sort_by_key(|r| {
        sort_operations.push(value);
        r.width
    });
    println!("{:#?}", list);
}
```



```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut num_sort_operations = 0;
    list.sort_by_key(|r| {
        num_sort_operations += 1;
        r.width
    });
    println!("{:#?}, sorted in {num_sort_operations} operations", list);
}
```

Iteratori

- Omogućava vam da izvršite neki zadatak na nizu stavki
- Odgovoran je za logiku ponavljanja svake stavke i određivanje kada je sekvenca završena.
- *Lazy* - nemaju efekat dok ne pozovete metode koje treba da ih iskoriste

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();
```

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    println!("Got: {}", val);
}
```

Trait Iterator i metoda *next*

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // methods with default implementations elided
}
```

Poziv metode *next*

```
#[test]
    fn iterator_demonstration() {
        let v1 = vec![1, 2, 3];

        let mut v1_iter = v1.iter();

        assert_eq!(v1_iter.next(), Some(&1));
        assert_eq!(v1_iter.next(), Some(&2));
        assert_eq!(v1_iter.next(), Some(&3));
        assert_eq!(v1_iter.next(), None);
    }
```

Upotreba iteratora

- Adapteri - metode koje pozivaju metodu *next*

```
#[test]
    fn iterator_sum() {
        let v1 = vec![1, 2, 3];

        let v1_iter = v1.iter();

        let total: i32 = v1_iter.sum();

        assert_eq!(total, 6);
    }
```

Metode koje proizvode iterator

- Adapteri iteratora - metode koje proizvode iteratore, tako što menjaju neki aspekt orginalnog iteratora

```rust
let v1: Vec<i32> = vec![1, 2, 3];

v1.iter().map(|x| x + 1);
```

```rust
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

Zadaci

1. Iz fajla učitati brojeve, a zatim:
   a. Izračunati zbir brojeva upotrebom metode *sum*
   b. Prikazati sve parne brojeve upotrebom metode *filter*
   c. Svaki broj kvadrirati, a zatim prikazati dobijeni rezultat
2. Napraviti vektor stringova, a zatim ga sortirati u opadajućem redosledu.