

Rust: Closure, Iterator, Generic, Lifetime

Hafta - 4

Closure

Diğer dillerdeki "Lambda fonksiyon."

Closure

```
fn bir_ekle(x:u32) -> u32 {  
    x+1  
}
```

Normal bir fonksiyon tanımı

Closure

```
let bir_ekle = |x:u32| -> u32 {  
    x+1  
};
```

Closure fonksiyon tanımı

Closure

```
let bir_ekle = |x:u32| -> u32 { x+1 };
```

Daha kısa yazılabilir

Closure

```
let bir_ekle = |x| { x+1 };
```

Daha da kısa yazılabilir

Closure

```
let bir_ekle = |x| x+1;
```

Closure

```
let bir_ekle = |x| x+1;  
println!("{}", bir_ekle(9));
```

> 10

Closure

```
let selamla = || println!("Selam");  
selamla();
```

> Selam

Closure

```
let isim = String::from("Emin");  
  
let selamla = |i| println!("Selam {}!", i);  
  
selamla(isim);  
  
> Selam Emin!
```

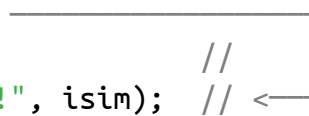
Closure

```
let isim = String::from("Emin");  
  
let selamla = |i| println!("Selam {}", i);  
  
selamla(isim);  
selamla(5);
```

```
ef@monster-pardus:~/Belgeler/Rust/playground$ rustc main.rs  
error[E0308]: mismatched types  
  --> main.rs:32:10  
32 |         selamla(5);  
   |         ^- help: try using a conversion method: `.to_string()`  
   |         expected struct `String`, found integer  
  
error: aborting due to previous error  
  
For more information about this error, try `rustc --explain E0308`.  
ef@monster-pardus:~/Belgeler/Rust/playground$
```

"i" parametresinin tipini derleyici "`i: String`" diye anladı ve bizim yerimize tipi atadı.
Closure fonksiyonlarındaki parametrelerin tipi de tıpkı fonksiyon tipleri gibi sadece belirli bir tiptir.
Dolayısıyla sadece o tipteki parametreler verilebilir.

Closure

```
let isim = String::from("Emin"); //  // "isim"i scope'tan okur  
let selamla = || println!("Selam {}!", isim); // <—  
selamla();
```

> Selam Emin!

Closure

```
let isim = String::from("Emin"); // ----- move -----  
                                // "isim"i scope'tan closure'un içine taşı  
let selamla = move || println!("Selam {}!", isim); // <-----  
  
selamla();  
  
isim; // Burada erişilebilir değil.
```

```
ef@monster-pardus:~/Belgeler/Rust/playground$ rustc main.rs  
error[E0382]: use of moved value: `isim`  
--> main.rs:7:1  
2 | let isim = String::from("Emin");  
   |     ---- move occurs because `isim` has type `String`, which does not implement the `Copy` trait  
3 |  
4 | let selamla = move || println!("Selam {}!", isim);  
   |               -----  
   |               |  
   |               value moved into closure here  
...  
7 | isim;  
   | ^^^^ value used here after move  
  
error: aborting due to previous error  
  
For more information about this error, try `rustc --explain E0382`.  
ef@monster-pardus:~/Belgeler/Rust/playground$
```

Iterator

Iterator'lar bir seri üzerinde sırayla işlem yapmak için kullanılır.

Trait `std::iter::Iterator` 

1.0.0 [-][src]

```
pub trait Iterator {  
    type Item;  
    [-] fn next(&mut self) -> Option<Self::Item>;  
  
    fn size_hint(&self) -> (usize, Option<usize>) { ... }  
    fn count(self) -> usize { ... }  
    fn last(self) -> Option<Self::Item> { ... }  
    fn advance_by(&mut self, n: usize) -> Result<(), usize> { ... }  
    fn nth(&mut self, n: usize) -> Option<Self::Item> { ... }  
    fn step_by(self, step: usize) -> StepBy<Self> ① { ... }  
    fn chain<U>(self, other: U) -> Chain<Self, <U as IntoIterator>::IntoIter> ①  
    where  
        U: IntoIterator<Item = Self::Item>,  
    { ... }  
    fn zip<U>(self, other: U) -> Zip<Self, <U as IntoIterator>::IntoIter> ①
```

Iterator

```
pub trait Iterator {  
    type Item;  
  
    // Bir sonraki item varsa Some ile döndürür. Yoksa None.  
    fn next(&mut self) -> Option<Self::Item>;  
  
    // ...  
}
```

```
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);    // Bir sonraki eleman yok.
```

Iterator

```
let v1 = vec![1, 2, 3];  
  
v1.iter().map(|x| x + 1);
```

```
|      v1.iter().map(|x| x + 1);  
|      ^^^^^^^^^^^^^^^^^^^^^^^^^  
|  
= note: iterators are lazy and do nothing unless consumed
```

Iterator'lar kullanılmadıkça bir şey yapmazlar.

Iterator

```
let v1 = vec![1, 2, 3];  
  
let v2: Vec<i32> = v1.iter().map(|x| x + 1).collect();  
  
println!("{:?}", v2);
```

> [2, 3, 4]

`map(|x| x+1)` iterator'daki her elemana +1 işlemi uygulanmasını söyler.

Kendi Iterator'umuz: CiftSayici

```
struct CiftSayici {  
    count: u32,  
}  
  
impl CiftSayici {  
    fn new() -> Self {  
        Self { count: 0 }  
    }  
}
```

Kendi Iterator'umuz: CiftSayici

```
// Iterator traitini CiftSayici için implement ediyoruz.  
impl Iterator for CiftSayici {  
    type Item = u32;  
  
    fn next(&mut self) -> Option<Self::Item> {  
        self.count += 2;  
  
        Some(self.count)  
    }  
}
```

Kendi Iterator'umuz: CiftSayici

```
impl Iterator for CiftSayici {  
    type Item = u32;  
  
    fn next(&mut self) -> Option<Self::Item> {  
        self.count += 2;  
  
        Some(self.count)  
    }  
}
```

```
let mut sayici = CiftSayici::new();  
  
assert_eq!(sayici.next(), Some(2));  
assert_eq!(sayici.next(), Some(4));  
assert_eq!(sayici.next(), Some(6));  
assert_eq!(sayici.next(), Some(8));  
assert_eq!(sayici.next(), Some(10));  
  
// Sayı u32 limitine ulaşana kadar gidebiliriz.
```

Kendi Iterator'umuz: CiftSayici

```
impl Iterator for CiftSayici {  
    type Item = u32;  
  
    fn next(&mut self) -> Option<Self::Item> {  
        // 6'dan küçük olarak limitledik  
        if self.count < 6 {  
            self.count += 2;  
  
            Some(self.count)  
        } else {  
            None  
        }  
    }  
}
```

```
let mut sayici = CiftSayici::new();  
  
assert_eq!(sayici.next(), Some(2));  
assert_eq!(sayici.next(), Some(4));  
assert_eq!(sayici.next(), Some(6));  
assert_eq!(sayici.next(), None);
```

Iterator

Kullanışlı iterator fonksiyonları

- `map()` -> Iterator'ün bütün elemanlarına işlem yapmak için
- `filter()` -> Iterator'ün elemanlarını belirli bir koşula göre filtrelemek için
- `nth()` -> Iterator'ün n'inci elemanına erişmek için
- `count()` -> Iterator'ün boyutuna erişmek için
- `enumerate()` -> Iterator'ün elemanlarını indisleriyle beraber kullanmak için

<https://doc.rust-lang.org/std/iter/trait.Iterator.html>

iterator.map()

```
impl Iterator for CiftSayici {  
    type Item = u32;  
  
    fn next(&mut self) -> Option<Self::Item> {  
        if self.count < 6 { // limitedik  
            self.count += 2;  
  
            Some(self.count)  
        } else {  
            None  
        }  
    }  
}
```

```
let sayici = CiftSayici::new();  
  
let tekler:Vec<u32> = sayici.map(|x| x-1).collect();  
  
println!("{:?}", tekler);
```

> [1, 3, 5]

```
[~] fn map<B, F>(self, f: F) -> Map<Self, F> ⓘ [src]  
    where  
        F: FnMut(Self::Item) -> B,
```

Takes a closure and creates an iterator which calls that closure on each element.

`map()` transforms one iterator into another, by means of its argument: something that implements `FnMut`. It produces a new iterator which calls this closure on each element of the original iterator.

iterator.filter()

```
impl Iterator for CiftSayici {  
    type Item = u32;  
  
    fn next(&mut self) -> Option<Self::Item> {  
        if self.count < 6 { // limitedik  
            self.count += 2;  
  
            Some(self.count)  
        } else {  
            None  
        }  
    }  
}
```

```
let sayici = CiftSayici::new();  
  
let filtreli:Vec<u32> = sayici.filter(|&x| x < 5).collect();  
  
println!("{:?}", filtreli);
```

> [2, 4]

```
[1] fn filter<P>(self, predicate: P) -> Filter<Self, P> ⓘ [src]  
    where  
        P: FnMut(&Self::Item) -> bool,
```

Creates an iterator which uses a closure to determine if an element should be yielded.

Given an element the closure must return `true` or `false`. The returned iterator will yield only the elements for which the closure returns true.

iterator.nth()

```
impl Iterator for CiftSayici {  
    type Item = u32;  
  
    fn next(&mut self) -> Option<Self::Item> {  
        if self.count < 6 { // limitledik  
            self.count += 2;  
  
            Some(self.count)  
        } else {  
            None  
        }  
    }  
}
```

```
let sayici = CiftSayici::new();  
  
let ucuncu:Option<u32> = sayici.nth(2);  
  
println!("{:?}", ucuncu);
```

> Some(6)

```
[1] fn nth(&mut self, n: usize) -> Option<Self::Item> [src]
```

Returns the `nth` element of the iterator.

Like most indexing operations, the count starts from zero, so `nth(0)` returns the first value, `nth(1)` the second, and so on.

Note that all preceding elements, as well as the returned element, will be consumed from the iterator. That means that the preceding elements will be discarded, and also that calling `nth(0)` multiple times on the same iterator will return different elements.

`nth()` will return `None` if `n` is greater than or equal to the length of the iterator.

iterator.enumerate()

```
impl Iterator for CiftSayici {  
    type Item = u32;  
  
    fn next(&mut self) -> Option<Self::Item> {  
        if self.count < 6 { // limitledik  
            self.count += 2;  
  
            Some(self.count)  
        } else {  
            None  
        }  
    }  
}
```

```
let sayici = CiftSayici::new();  
  
for (indis, eleman) in sayici.enumerate() {  
    println!("[{}] = {}", indis, eleman);  
}
```

```
> [0] = 2  
> [1] = 4  
> [2] = 6
```

```
[1] fn enumerate(self) -> Enumerate<Self> ⓘ [src]
```

Creates an iterator which gives the current iteration count as well as the next value.

The iterator returned yields pairs `(i, val)`, where `i` is the current index of iteration and `val` is the value returned by the iterator.

`enumerate()` keeps its count as a `usize`. If you want to count by a different sized integer, the `zip` function provides similar functionality.

iterator.enumerate()

```
let v1 = vec![5, 10, 15];  
  
for (indis, eleman) in v1.iter().enumerate() {  
    println!("v1[{}] = {}", indis, eleman);  
}
```

```
> v1[0] = 5  
> v1[1] = 10  
> v1[2] = 15
```

Örnek: Fibonacci hesaplayan iterator

```
struct Fibonacci {  
    first:u32,  
    second:u32  
}  
  
impl Fibonacci {  
    fn new() -> Self {  
        Self {first: 0, second: 1}  
    }  
}  
  
impl Iterator for Fibonacci {  
    type Item = u32;  
  
    fn next(&mut self) -> Option<Self::Item> {  
        let new_first = self.second;  
        let new_second = self.first + self.second;  
  
        self.first = new_first;  
        self.second = new_second;  
  
        Some(new_second)  
    }  
}
```

Örnek: Fibonacci hesaplayan iterator

```
struct Fibonacci {  
    first:u32,  
    second:u32  
}  
  
impl Fibonacci {  
    fn new() -> Self {  
        Self {first: 0, second: 1}  
    }  
}  
  
impl Iterator for Fibonacci {  
    type Item = u32;  
  
    fn next(&mut self) -> Option<Self::Item> {  
        let new_first = self.second;  
        let new_second = self.first + self.second;  
  
        self.first = new_first;  
        self.second = new_second;  
  
        Some(new_second)  
    }  
}
```

```
fn main() {  
    let mut f = Fibonacci::new();  
  
    println!("{}", f.next().unwrap() );  
    println!("{}", f.next().unwrap() );  
    println!("{}", f.next().unwrap() );  
    println!("{}", f.next().unwrap() );  
    println!("{}", f.next().unwrap() );  
    println!("{}", f.next().unwrap() );  
}
```

```
> 1  
> 2  
> 3  
> 5  
> 8  
> 13
```

Generic

Genel tipleme.
Aynı kodda birden fazla tipi ele alabilmek için.

Generic

```
fn largest_i32(list: &[i32]) -> i32 {  
    let mut largest = list[0];  
  
    for &item in list {  
        if item > largest {  
            largest = item;  
        }  
    }  
  
    largest  
}  
  
fn largest_char(list: &[char]) -> char {  
    let mut largest = list[0];  
  
    for &item in list {  
        if item > largest {  
            largest = item;  
        }  
    }  
  
    largest  
}
```

```
fn largest<T>(list: &[T]) -> T {  
    let mut largest = list[0];  
  
    for &item in list {  
        if item > largest {  
            largest = item;  
        }  
    }  
  
    largest  
}
```

Generic

```
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let char_list = vec!['y', 'm', 'a', 'x'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

```
ef@monster-pardus:~/Belgeler/Rust/playground$ rustc main.rs
error[E0369]: binary operation `>` cannot be applied to type `T`
  --> main.rs:30:17
30 |         if item > largest {
    |                ^ ----- T
    |                |
    |                T

help: consider restricting type parameter `T`
26 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> T {
    |             +++++++++++++++++++++

error: aborting due to previous error

For more information about this error, try `rustc --explain E0369`.
ef@monster-pardus:~/Belgeler/Rust/playground$
```

`largest<T>` fonksiyonu her tipe açık fakat `item > largest` işlemi her tipe uygulanamıyor.

Bize `std::cmp::PartialOrd` traitini implement eden bir tip filtresiyle fonksiyonu sadece belirli tiplere kısıtlamamızı öneriyor.

Yani sadece büyüktür küçüktür gibi kıyaslama yapma kodunu implement etmiş türler geçerli olacak.

Generic

```
fn largest<T: PartialOrd>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let char_list = vec!['y', 'm', 'a', 'x'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

`PartialOrd` ile `item > largest` işleminin gelecek her tip için sağlandığını garantiledik.

Şimdi ise kopyalama işlemi olmayan tiplerde çıkan move hatasıyla

```
let mut largest = list[0];
```

`largest` değişkeni listedeki elemanın sahipliğini almaya çalışıyor.

```
ef@monster-pardus:~/Belgeler/Rust/playground$ rustc main.rs
error[E0508]: cannot move out of type `[T]`, a non-copy slice
  --> main.rs:27:23
27 |         let mut largest = list[0];
   |                         ^^^^^^^
   |                         |
   |                         cannot move out of here
   |                         move occurs because `[T]` has type `[T]`, which does not implement the `Copy` trait
   |                         help: consider borrowing here: `&list[0]`

error[E0507]: cannot move out of a shared reference
  --> main.rs:29:18
29 |         for &item in list {
   |             ^^^^^
   |             ||
   |             |data moved here
   |             |move occurs because `item` has type `T`, which does not implement the `Copy` trait
   |             help: consider removing the `&`: `item`

error: aborting due to 2 previous errors
```

Generic

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {  
    let mut largest = list[0];    // <-- Copy işlemi  
  
    for &item in list {  
        if item > largest {        // <-- PartialOrd işlemi  
            largest = item;        // <-- Copy işlemi  
        }  
    }  
  
    largest  
}  
  
fn main() {  
    let char_list = vec!['y', 'm', 'a', 'x'];  
  
    let result = largest(&char_list);  
    println!("The largest char is {}", result);  
}
```

> The largest char is y

Sahiplik taşıma hatası vermemesi için **Copy** traitini implement etme filtresi de getirdik.

Böylece `largest = list[0]` sahiplik almak yerine `list[0]`'ın değerini kopyalayacak.

Generic

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {  
    // ...  
}
```



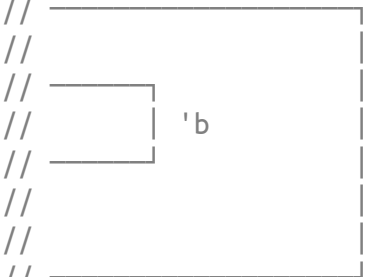

```
fn largest<T>(list: &[T]) -> T  
where T: PartialOrd + Copy  
{  
    // ...  
}
```

Generic tiplerin filtrelerini **where** ile ayrı bir satırda da yazabiliriz.

Lifetime

Referans deęiřkenlerinin yařam süresi.

Lifetime

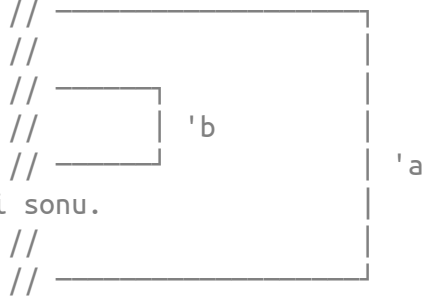
```
fn main() {  
    let r;           //  'a'  
    {               //  'b'  
        let x = 1;  
        r = &x;  
    }  
    println!("{}", r);  
}
```

r değişkeni x'ten daha **uzun** yaşadığı için onu **referans edemez**.

```
ef@monster-pardus:~/Belgeler/Rust/playground$ rustc diger.rs  
error[E0597]: `x` does not live long enough  
--> diger.rs:5:7  
5 |         r = &x;  
   |         ^^ borrowed value does not live long enough  
6 |     }  
   |     - `x` dropped here while still borrowed  
7 |  
8 |     println!("{}", r);  
   |                   - borrow later used here  
  
error: aborting due to previous error
```

Lifetime

```
fn main() {  
    let x = 5;           //   
                          //   
    let r = &x;          //   
                          //   
    println!("{}", r);  // 'b  
    // r'nin yaşam süresi sonu.   
                          //   
                          //   
}
```



> 5

r değişkeni x'ten daha **kısa** yaşadığı için onu **referans** edebilir.

Lifetime

Rust'ta referansların siz açıkça yazmasanız da yaşam süreleri implicit olarak mevcuttur.
Explicit olarak da yazılması gereken durumlar da vardır.

```
&i32          // i32 bir değerin referansı  
&'a i32       // Yaşam süresi(lifetime) açıkça(explicit) belirtilmiş bir referans  
&'a mut i32   // Yaşam süresi açıkça belirtilmiş ve mutable(değiştirilebilir) bir referans
```

Lifetime

```
fn longest(x: &str, y: &str) -> &str {  
    if x.len() > y.len() {  
        x  
    } else {  
        y  
    }  
}  
  
fn main() {  
    let a = "selam";  
    let b:&'static str = "bu daha uzun";  
    longest(a, b);  
}
```

x veya y'nin yaşam süreleri bilinmediği için geçerli olup olmadıklarını da bilmiyor.

```
ef@monster-pardus:~/Belgeler/Rust/playground$ rustc diger.rs  
error[E0106]: missing lifetime specifier  
--> diger.rs:1:33  
1 | fn longest(x: &str, y: &str) -> &str {  
  |               ----      ----      ^ expected named lifetime parameter  
  |  
  = help: this function's return type contains a borrowed value, but the signature  
        does not say whether it is borrowed from `x` or `y`  
  help: consider introducing a named lifetime parameter  
1 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
  |               +++++  ++          ++          ++  
  
error: aborting due to previous error  
  
For more information about this error, try `rustc --explain E0106`.
```


Lifetime

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
    if x.len() > y.len() {  
        x  
    } else {  
        y  
    }  
}  
  
fn main() {  
    let a = "selam";  
    let b:&'static str = "bu daha uzun";  
    println!("{}", longest(a, b));  
}
```

> bu daha uzun

x ve y referanslarına aynı yaşam süresi etiketi verdik.

Yani ikisi de ortak olarak geçerli olduğu bir yaşam süresine sahip olmalı.

Bir sonraki slayttaki örnekle daha iyi anlaşılabilir.

Lifetime

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
    if x.len() > y.len() {  
        x  
    } else {  
        y  
    }  
}
```

İkisi de ortak zamanda geçerli olması gerektiği için derleyici hata verdi.

Çünkü `string2`'nin yaşam süresi `scope`'tan sonra bitiyor.

Dolayısıyla `longest` fonksiyonu `string2.as_str()` referansını uzun diye return ederse aşağıdaki `println!` kodu geçersiz olacak.

```
fn main() {  
    let string1 = String::from("bu daha uzun");  
    let result;  
  
    {  
        let string2 = String::from("kısa");  
        result = longest(string1.as_str(), string2.as_str());  
    }  
  
    println!("{}", result);  
}
```

```
ef@monster-pardus:~/Belgeler/Rust/playground$ rustc diger.rs  
error[E0597]: `string2` does not live long enough  
  --> diger.rs:15:44  
15 |         result = longest(string1.as_str(), string2.as_str());  
   |                                     ^^^^^^^ borrowed value does not  
live long enough  
16 |     }  
   |     - `string2` dropped here while still borrowed  
17 |  
18 |     println!("{}", result);  
   |                   ----- borrow later used here  
  
error: aborting due to previous error  
  
For more information about this error, try `rustc --explain E0597`.  
ef@monster-pardus:~/Belgeler/Rust/playground$
```

Homeworks

1. Filter even numbers on a given list
2. Multiply all numbers with 2 on a given list
3. Implement an iterator MultiFibonacci which implements this algorithm: $f(n) = f(n-1) * f(n-2)$ and $n > 0$