RUST
Lifetimes (yaşam süresi)
'a

Dangling References
Borrow Checker
Struct & Lifetime
Function & Lifetime

Lifetimes Nedir?

Yaşam süresi olarak tercüme edebiliriz

Generic tiplerinin farklı bir türüdür

Her referans tipinin bir yaşam süresi (geçerli olduğu kapsam) vardır.

Lifetime'ı bir referans tipinin geçerlilik süresini belirtmek için kullanırız.

Lifetime'ın ana amacı dangling references (referansların sarkması yani referansların geçerliliğini kaybetmesi) durumunu önlemektir.

Lifetime Annotation Sözdizimi

Lifetime referanslardan herhangi birinin ne kadar süreyle yaşayacağını değiştirmez.

Birden fazla referansın yaşam sürelerinin birbirleriyle olan ilişkilerini, yaşam sürelerini etkilemeden tanımlarlar.

Fonksiyon tanımlarken generic veri tipini <T> kullanarak, nasıl ki herhangi bir veri tipi fonksiyona gönderilebiliyorsa; generic lifetime parametresi kullanarak ta herhangi bir yaşam süresi olan referansları (başvuruları) fonksiyona gönderebiliriz.

Lifetime Annotation Sözdizimi

Lifetime isimleri ;

' (kesme) işareti ile başlar, kısa tutulur ve küçük harf kullanılır

Genellikle ilk lifetime için **'a** kullanılır.

Bu, **& simgesinden sonra** yazılır ve **veri türü ile aralarına boşluk** eklenir.

```
&i32
& 'a i32
& 'a mut i32
```

Dangling References - Referans Sarkmaları

Bir referansın gösterdiği değişkenin, kapsam dışına çıkmasından dolayı, referansın gösterdiği değişkenin geçersiz hale gelmesidir.

```
fn main() {

let r;

{
        let x = 5;
        r = &x;

}

println!("r: {}", r);   // HATA OLUŞUR

}
```

x değişkeni kapsam dışına çıkınca geçersiz olduğu için r'ye atanan referans değeri de geçersiz olur ve hata oluşur.

Bu duruma dangling references olarak isimlendirilir.

Borrow Checker - Ödünç Alma Denetleyicisi

Rust, ödünç alınan değerlerin geçerli olduğunu kontrol etmek için kapsamları karşılaştırır.

```
fn main() {

let r;                          // ----------+--'a
                                //           |
{                               //           |
        let x = 5;              // -+-- 'b   |
        r = &x;                 //  |        |
}                               // -+        |
                                //           |
println!("r: {}", r);           //           |
}                               // ----------+
```

'a → r'nin yaşam süresidir
'b → x'in yaşam süresidir

'a > 'b

HATANIN GİDERİLMESİ

```
fn main() {

        let x = 5;                      // ----------+--'b
                                        //           |
        let r = &x;                     // -+-- 'a   |
                                        //  |        |
                                        //  |        |
        println!("r: {}", r);           //  |        |
                                        // -+        |
}                                       // ----------+
```

'a → r'nin yaşam süresidir
'b → x'in yaşam süresidir

'b > 'a

x'in yaşam süresi 'b, r'nin yaşam süresi 'a dan büyük olduğu için referans sürekli geçerli olur.

Fonksiyonlarda Yaşam Süreleri

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

```
fn longest(x: &str) -> &str {
    x
} // Burada belli olduğu için sorun
olmaz
```

Fonksiyona aktarılacak somut değerleri bilmiyoruz. Bu yüzden, burada döndürülen referansın x'e mi yoksa y'ye mi referans döndüreceği belli değildir.

Referanslar aynı kapsamda olmadığı için borrow checker yaşam sürelerini kontrol edemez.

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

Fonksiyona aynı lifetime'a sahip referanslar gelecek ve aynı lifetime'a sahip bir referans dönecektir.
Burada yaşam sürelerini değiştirmiyoruz.
Borrow checker'ın bu kısıtlamaya uymayan değerleri reddetmesi gerektiğini belirtiyoruz.
Fonksiyonun x ve y nin yaşam süresini bilmesine gerek yoktur.

```rust
fn main() {
        let string1 = String::from("abcd");
        let string2 = "xyz";
        let result = longest(string1.as_str(), string2);
        println!("The longest string is {}", result);
}
```

Generic lifetime parametreleri ekleyerek, referanslar arasında ilişkiyi belirlememiz gerekir.

Bu sayede borrow checker referansları analiz edebilir.
Bu da hatayı giderir.

'a yaşam süresi;

x ve y'nin ömürlerinden küçük olanına eşit olan somut yaşam süresini alır.

Dönen referansa aynı yaşam süresi parametresi 'a ile açıklama eklediğimiz için, dönen referans x ve y'nin yaşam sürelerinden küçük olanının uzunluğu için de geçerli olacaktır.

Yani farklı kapsamlardaki referansları gönderebiliriz, aşağıdaki gibi.

```
fn main() {
      let string1 = String::from("long string is long");
      {
            let string2 = String::from("xyz");
            let result = longest(string1.as_str(),
string2.as_str());
            println!("The longest string is {}", result);
      }
}
```

Buradaki kodumuz çalışacaktır.

Burada farklı lifetime'a ait veriler fonksiyona gönderilmektedir.

```
fn main() {
      let string1 = String::from("long string is long");
      let result;

      {
            let string2 = String::from("xyz");
            result = longest(string1.as_str(),
string2.as_str());
      }
      // HATA OLUŞUR
      println!("The longest string is {}", result); }
```

Döndürülen referansın yaşam süresinin, aktarılan referansların yaşam sürelerinden küçük olanıyla aynı olduğunu söyledik. Bu yüzden ödünç denetleyicisi bunu onaylamaz

Yani string2 kapsam dışına çıktığı için result değişkeninin değeri geçersiz olur. Çünkü result değişkeninin değeri referanstır.

Fonksiyon Yapısına Göre Lifetime Kullanımı

Yaşam süresi parametrelerini belirtme şekliniz, fonksiyonunuzun ne yaptığına bağlıdır.

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

Sadece ilk parameter geri döndürüldüğü için y'ye life time eklemeye gerek yoktur.

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
      let result = String::from("really long string");
      result.as_str()
}
// HATA OLUŞUR
```

Dönen değerin lifetime ile ilgilisi olmadığından hata oluşur. Çünkü scope tamamlandığı için result geçersiz olur.

Struct ve Lifetime

Referans tutan struct tanımladığımızda, struct tanımlamasına da lifetime tanımlaması eklememiz gerekir. Generic types gibi!

```
struct ÖnemliAlıntı<'a> {
    kısım: &'a str,
}
// Bir romandan belli bir kısmı olamak için tanımlanmış
bir örnektir.
```

Bu tanımlama;

Struct örneğinin (struct'tan oluşturulan değişken), lifetime ile tanımlanan struct alanından (yani kısım değişkeninden) daha uzun bir lifetime'a sahip olamayacağı anlamına gelir.

```
fn main() {
	let roman = String::from("O mavi kuştan, yanar döner kuştan.
Hani, su kıyılarındaki yarları…");

	let ilk_cümle = roman.split('.').next().expect("'.'
bulunamadı!");

	let alıntı = ÖnemliAlıntı {
		kısım: ilk_cümle,
	};
}
```

alıntı struct örneği, sahibi **roman** değişkeni olan String'in ilk cümlesinin referansını tutmaktadır. Roman değişkenindeki veri ÖnemliAlıntı'nın örneğinden önce oluşturulmaktadır. Struct kapsam dışına çıkmadan **roman** değişkeni kapsam dışına çıkmayacağı için referans geçerlidir.

Yani kısım ile, referans gösterdiği değer aynı lifetime süresine sahiptir.

Lifetime Elision Kuralları

Rust'ın referans tiplerini analiz etmek için sahip olduğu kalıplara "lifetime elision" kuralları denir.

Fonksiyon ve methodlarda kullanılan parametrelere "input lifetimes", geri dönen değerlerde kullanılan lifetime lara "output lifetimes" olarak isimlendirilmektedir.

Lifetime Elision Kuralları

Bu kuralları programcı kullanmaz, Rust bu kuralları kendi içinde uygular.

1 - Derleyici, referans olan her parametre için bir yaşam süresi parametresi atar. Başka bir deyişle, bir parametresi olan bir fonksiyon sadece bir tane lifetime parametresi alır.

```
fn foo<'a>(x: &'a i32);
```

iki parametresi olan bir fonksiyon iki ayrı yaşam süresi parametresi alır

```
fn foo<'a, 'b>(x: &'a i32, y: &'b i32);
```

Bu böyle devam eder.

Lifetime Elision Kuralları

2 - Bir tane input lifetime parametresi varsa, bu parametre tüm output lifetime parametrelerine uygulanır.

```
fn foo<'a>(x: &'a i32) -> &'a i32
```

Lifetime Elision Kuralları

3 - Birden fazla input lifetime parametresi varsa fakat bunlardan,

biri **&self** veya **&mut self** ise (çünkü bu bir metottur)

self'in lifetime'ı tüm output çıktı parametrelerine uygulanır.

```rust
fn first_word(s: &str) -> &str {
        let bytes = s.as_bytes();
        for (i, &item) in bytes.iter().enumerate() {
                if item == b' ' {
                        return &s[0..i];
                }
        }
        &s[..]
}

fn main() {
        let my_string = String::from("hello world");
        // first_word works on slices of `String`s
        let word = first_word(&my_string[..]);

        let my_string_literal = "hello world";
        // first_word works on slices of string literals
        let word = first_word(&my_string_literal[..]);
        // Because string literals *are* string slices already,
        // this works too, without the slice syntax!
        let word = first_word(my_string_literal); }
```

Derleyici first_Word() fonsiyonu için aşağıdaki gibi lifetime'lar ekler.

```
fn first_word(s: &str) -> &str {
```

```
fn first_word<'a>(s: &'a str) -> &str { // BİRİNCİ KURAL
```

```
fn first_word<'a>(s: &'a str) -> &'a str { // İKİNCİ KURAL
```

Derleyici longest() fonsiyonu için aşağıdaki gibi lifetime'lar ekler. Fakat

```
fn longest(x: &str, y: &str) -> &str {
```

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

Birden fazla giriş parametresi olduğu için ikinci kural geçerli olmaz. Output lifetime parametresini belirleyemez.

Üçüncü kural da uygulanmaz

Bu yüzden de lifetime parametrelerini belirtmemiz gerekecektir.

Lifetime ve Metod Tanımlamaları

Metod tanımlarken, ilgili struct'a ait lifetime'ı kullanmamız gerekir.
Lifetime'ın nerede kullanacağımız amacımıza göre değişir.

```
impl<'a> ÖnemliAlıntı<'a> {

    fn seviye(&self) -> i32 {
        3
    }

}
```

```
impl<'a> ÖnemliAlıntı<'a> {

        fn duyuru_ve_donus_bölümü(&self, duyuru: &str) -> &str {

                println!("Lütfen dikkat : {}", duyuru);
                self.kısım

        }
}
```

Burada birinci ve üçüncü kural uygulanır.

&self ve duyuru'ya ayrı lifetime parametreleri eklenir.

Dönüş tipine &self'in lifetime parametresi ekllenir.

Static Lifetime - 'static

Özel bir lifetime'dır.
Referansın lifetime süresi, programın süresine eşit olur.
&str türleri static lifetime'a sahiptir. Çünkü programın binary kodunda bulunurlar.

```
let s: &'static str = "I have a static lifetime.";
```

Static veri tipleride static lifetime'a sahiptirler. (Ayrıca değiştirilebilirler)

static SAYI:i32 = 500;

'static lifetime bir referans için kullanmadan önce, referansın lifetime süresinin program boyunca devam edip etmediğini kontrol etmeniz gerekir.

Generic Type Parametleri, Trait Bounds ve Lifetime'ların Birlikte Kullanılması

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
        x: &'a str,
        y: &'a str,
        ann: T,
) -> &'a str where
        T: Display,
{
        println!("Announcement! {}", ann);
        if x.len() > y.len() {
                x
        } else {
                y
        }
}
```

```rust
fn main() {

        let string1 = String::from("abcd");
        let string2 = "xyz";

        let result = longest_with_an_announcement( string1.as_str(), string2,
"Today is someone's birthday!", );

        println!("The longest string is {}", result);
}
```

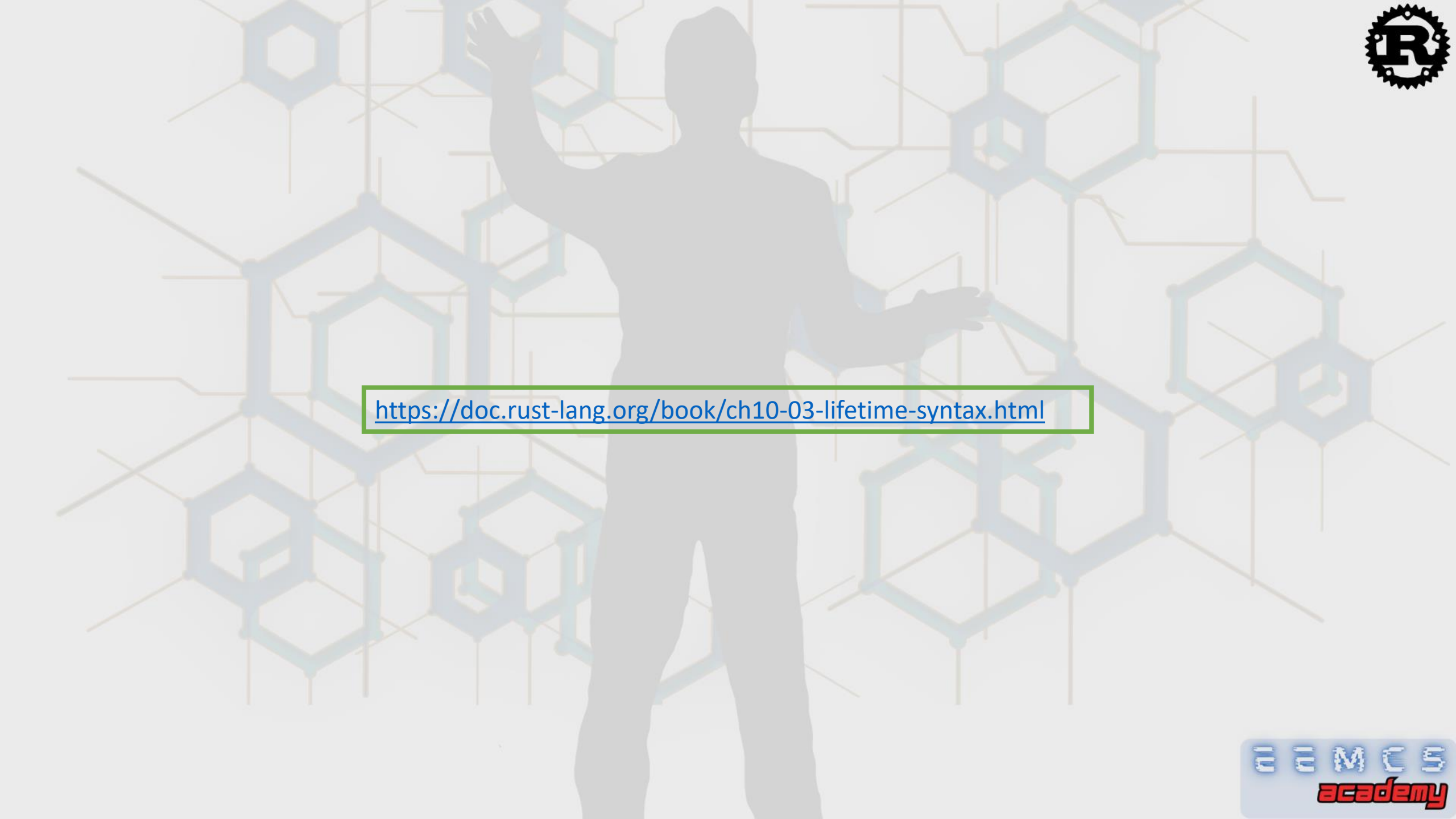
T genel veri tipi olduğu için, println ile yazdırmamız gerektiğinde Display trait'ini implement etmemiz gerekir.

where ifadesi T:Display trait'ini implement etmek için kullanılmıştır. Bunun yerine; fonksiyon tanımlamasında da implement edebilirdik.

```rust
fn longest_with_an_announcement<'a, T:Display ( veya
Debug )>(
```

ya da

Debug macrosunu da implement edebilirdik.



https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html

Celal AKSU
Bilişim Teknolojileri Öğretmeni

celalaksu@gmail.com

https://www.linkedin.com/in/cllaksu/

https://twitter.com/ksacll

https://www.youtube.com/@eemcs