


# Web Programlama I

## Ders 09: Selenium ve JaCoCo

---

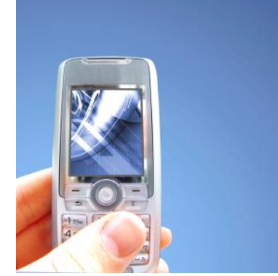
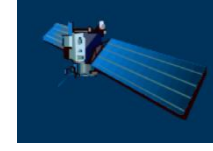



# Slaytlar Hakkında

- Bu slaytlar oldukça yüksek seviyede genel bir bakış açısı sunmaktadır
  - Detaylar Git repository'sindeki yorum satırlarında bulunmaktadır
-

# Yazılım Testi

# Yazılımlar her yerdedir!!! (yalnızca web uygulamaları yoktur...)





Yazılım  
uygulamaları  
yapmaları gerekeni  
yapıyor mu?

# Ariane 5 – ESA

---

- 4 Haziran 1996'da, Ariane 5 uçuşu başarısızlıkla sonuçlandı.
- \$500 milyon maliyet
- **Sebep: Yazılım Bug'ı**



# Ölümçöl Therac-25 Radyasyonu

1986'da, Texas'ta, bir kiři öldü



Kanada'da yaklaşık 50 milyon insan etkilendi

# 2003'te Elektrik Kesintisi





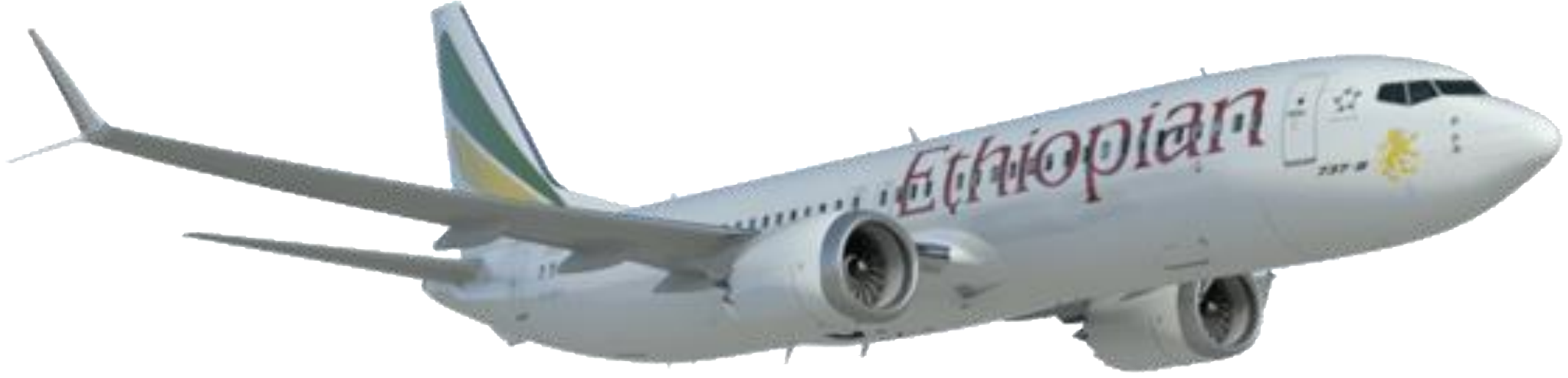


2010'da  
Toyota fren  
sistemindeki  
bug sebebiyle  
436,000 aracı  
geri çağırđı

Bir bug sebebiyle 45 dakikada \$460 milyon kayıp

# Knight Capital Group 2012





2019'da **yazılım problemi** sebebiyle Boeing 737 Max **kazası**; **157** insan öldü



# Ve böyle günlerce gidebilir...

2013 yılı itibarıyla, dünya  
çapında yazılım test  
maliyeti **\$312 milyardır**

2016'da, dünya çapında  
4.4 milyar insanı  
etkileyen ve **\$1.1 trilyona**  
mal olan 548 yazılım  
hatası kaydedilmiştir

Neden???

Ortak  
problem... Test  
yapılmaması!



# Yazılım Testi

- Yazılımlarda bug bulunur, ör: implementasyon hataları sebebiyle
- Bütün geliştiriciler bug oluştururlar, yalnızca öğrenciler değil, senior geliştiriciler bile
- Testler, müşteriye uygulamayı teslim etmeden önce hataların yerini bulmak için yardımcı olur
- Ancak, test hataların bulunmasını garanti etmez



# Manuel Test

- Geliştirdiğiniz uygulamada başlar
- Uygulamanın doğru çalışıp çalışmadığı kontrol edilir
  - ör, programın çökmemesi ve beklenen davranışın gerçekleşmesi beklenir
- Uygulamanın genel kullanım senaryoları denenir
- Ayrıca genel olmayan kullanım senaryoları da doğru çalışıp çalışmadığını kontrol etmek için denenir
- *Yazılım Tester'ı*: Uygulamanın yeni sürümü/özelligi yayınlanmadan önce uygulamayı test eden çalışandır



# Manuel Test Problemleri

- Pek çok insan sıkıcı bulmaktadır
  - Sistematik değildir: manuel süreç içerisinde bulunan önemli testler kaçırılabilir/unutulabilir
  - Pahalıdır: yazılım tester'ları da ailelerine bakabilmek için maaşa ihtiyaç duyar 😊
  - Her kod değişiminde yeni bir bug ortaya çıkabilir ve bir önceki gerçekleşen test geçersiz hale gelir
    - yani, tekrar tekrar tekrar test edilmelidir
  - Yine de, manuel bir test aşamasına ihtiyacın bulunmaktadır (örneğin, müşterilere büyük bir sürüm teslim etmeden önce), ancak manuel testin yanı sıra daha fazlasına da ihtiyacınız bulunmaktadır.
-





## Beta Test

- Biliyor olduğunuz manuel test türlerinden biridir. Özellikle video oyunlarda çokça karşınıza çıkar
- Yazılım, ücretsiz olarak test edecek bir grup potansiyel kullanıcıya "beta" durumunda yayınlanır!
- Online oyunlarda oldukça yaygındır, özellikle de ana sürümden önce sunucudaki yüksek yükleri test etmek için

# Test bağlama göre farklılık gösterir

- Eğer bir oyunda bug meydana gelirse kullanıcılar bir sonraki patch'i bekler...
  - Eğer ki lansman sırasında çok fazla hata varsa satışlar da düşebilir...
- Eğer bir enterprise sistemde (bir banka) bir bug varsa batmış olabilirsiniz...

# Otomatik Test

- Bir uygulamayı veya bölümlerini (ör. tek tek işlevler veya sınıflar) test etmek için otomatik olarak çalışan kod yazılabilir
- *Faydaları:*
  - Uygulamanın tek bir parçasını test edebilirsiniz
  - *Her kod değişiminde tekrar tekrar bütün testler çalıştırılabilir*

# Test Türleri

- *Birim Test*
    - En küçük birim (fonksiyon/sınıf) test edilir
  - *Entegrasyon Testi*
    - Farklı bileşenlerin bir araya geldiği senaryolar test edilir
  - *Sistem Testi*
    - Bütün uygulama test edilir, kullanıcının uygulama ile etkileşime geçebileceği arayüzler (ör, GUI) aracılığı ile test gerçekleştirilir
  - Ve daha bir sürü test türü...
    - Kullanıcı testi, güvenlik testi, performans testi, gürbüzlük testi vs...
-

# Java'da Test Koşma

- Temel framework *JUnit*'dir
  - TestNG ve Spock gibi farklı alternatifler de bulunmaktadır
- Yalnızca birim test değil, her türlü test için...
  - ör, Arquillian ve Spring servislerine entegrasyon testi yapılabilir

# Maven ve Test

- Maven build'in bir parçası olarak testleri çalıştırabilir
- Birim ve entegrasyon testleri arasında ayrım yapar
- Ancak bunlar SADECE isimlerdir ve birim ve entegrasyon testi kavramlarıyla ilgili DEĞİLDİR.
  - ör, mavende entegrasyon testi gibi birim testler de çalıştırılabilir
- *birim*: Surefire plugin'i ile çalıştırılır, isim kalıbı “*\*Test.java*”dır, “*mvn package*” aşamasından önce çalıştırılır (jar/war dosyası oluşturulmadan)
- *entegrasyon*: Failsafe plugin'i ile çalıştırılır, isim kalıbı “*\*IT.java*”dır, “*mvn package*” aşamasından sonra çalıştırılır (JAR/WAR dosyası oluşturulduktan sonra)

# Ne zaman “\*IT.java” kullanılır

- Eğer test JAR/WAR dosyasına ihtiyaç duyuyorsa
  - ör, mevcut built edilmiş uygulamayı çalıştırmak istediğinizde
- Eğer testiniz çok uzun sürede çalışıyorsa (ör., sistem testi), ve her severinde çalışmasını istemiyorsanız, yalnızca “*mvn package*” ile çalışmasını istiyorsanız
  - ör, bazı manuel testler için uygulamayı hızlı bir şekilde oluşturmak istiyorsunuz, ancak yine de hızlı birim testlerini çalıştırmak istiyorsunuz, her ihtimale karşı...



# Selenium







# Selenium

- Koddan bir tarayıcıyla etkileşim kurmanızı ve kontrol etmenizi sağlayan araç
  - Java ile yazılmıştır ancak pek çok dil için (C#/.Net gibi) kullanılabilir
  - Temel kullanım: web uygulamaları için önce sunucuyu başlatıp sonra butona tıklamak gibi browser üzerinden gerçek kullanıcı gibi gerçekleştirebileceğiniz sistem testi
-

# Selenium Gereksinimleri

- Java'da JAR kütüphanesini import etmek
- JUnit ile doğrudan çağırılabilir
- Bir tarayıcınız olmak **zorundadır** (ör., Chrome) ve Selenium *driver*'ı bu tarayıcıyı kullanır
  - Not: Docker'ı bir tarayıcı ve önceden yapılandırılmış driver'lar ile bir imajı çalıştırmak için de kullanabilir... ancak Docker imajının içinde çalışan tarayıcı GUI'sine doğrudan erişim olmadığı için debug daha zordur.
- Not: aynı testler farklı tarayıcılarda yürütülebilir

# HTML Etkileşimleri

- Selenium ile HTML bileşenleriyle etkileşime geçebilmek için önce onları seçebilmek gerekmektedir
- *En kolay yöntem* ise tıklamak veya bir şeyler yazmak istediğimiz HTML tag'ına *"id"* özelliği ekmektir
- Ancak bazen daha karmaşık sorgulara da ihtiyaç duyulabilir
  - ör, bir tabloda kaç adet satır olduğunu veya spesifik bir tag'ın kaç özelliği olduğunu hesaplamak için
  - Doğrulama için: ör., bir butona tıkladıktan sonra görüntülenen bir tabloya gerçekten yeni bir satır eklendiğini kontrol etmek için



# XPath

- HTML/XML dokümanlar için Query dilidir
    - [https://www.w3schools.com/xml/xpath\\_syntax.asp](https://www.w3schools.com/xml/xpath_syntax.asp)
  - Bir HTML/XML dokümanı tag'ların ve düğümlerin bir yol ile tanımlanabildiği ağaç yapısında temsil edilebilir
  - Bir yol ifadesi pek çok tag/düğüm ile eşleşebilir
  - XPath alternatifi “*CSS Selectors*”dür
-

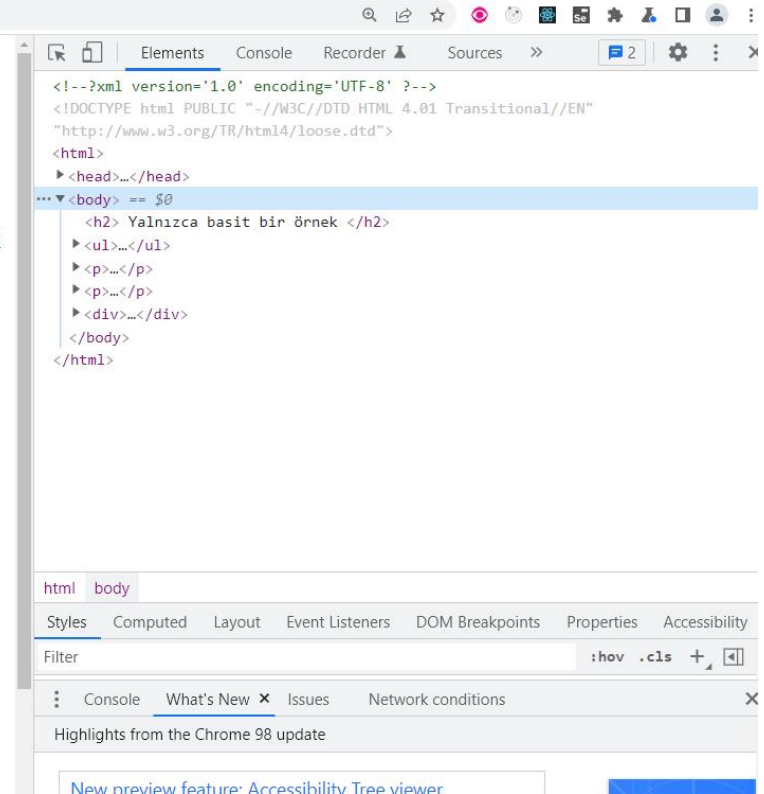
localhost:8080/base/home.html

## Yalnızca basit bir örnek

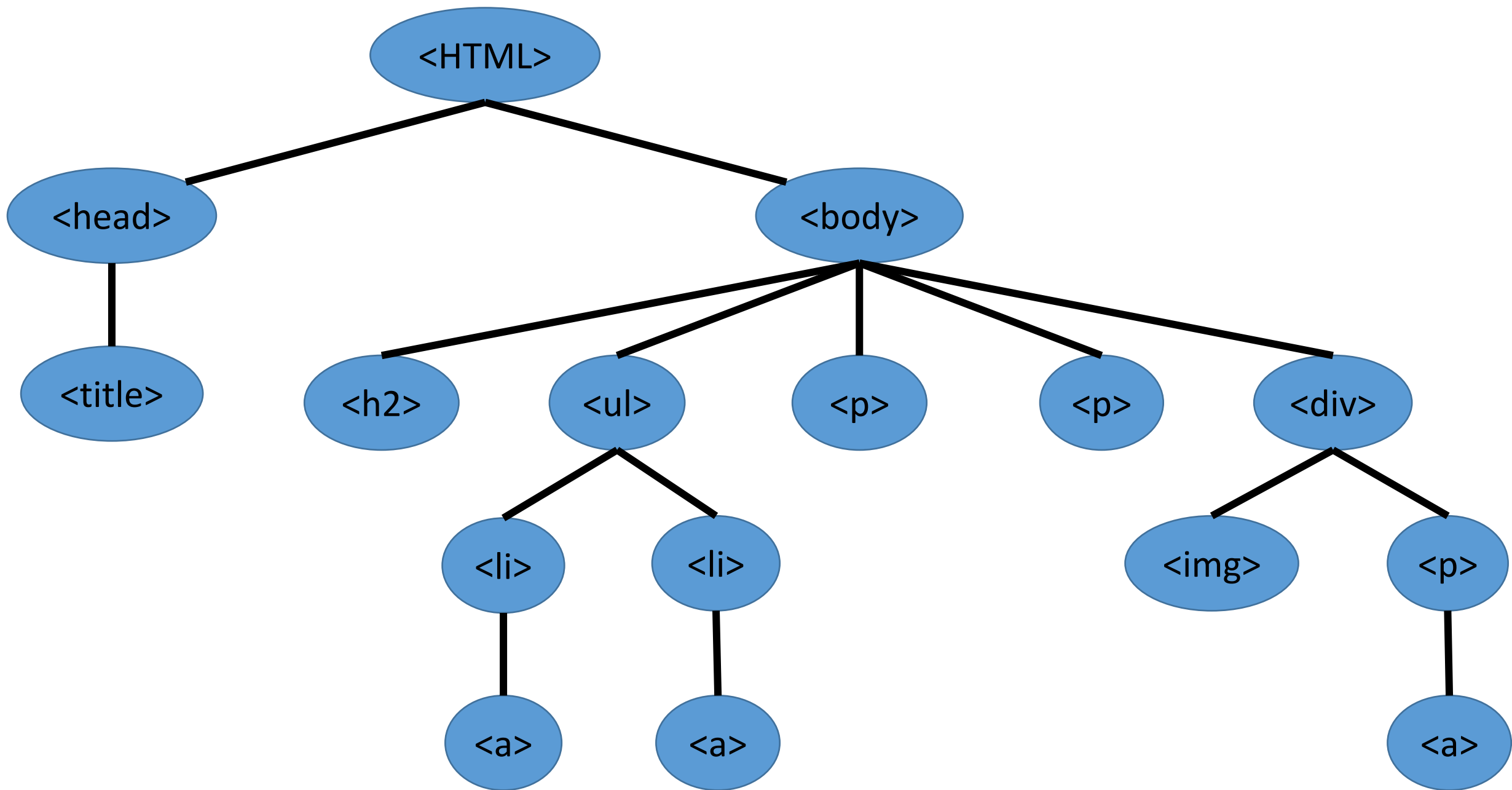
- webapp'teki farklı dosyalara bağlantı oluşturulabilir,örnek olarak: [foo/foo.html](#)
- Ancak WEB-INF altındaki dosyalar erişilemezdir, ör. [WEB-INF/web.xml](#) hata dönecektir

Bu uygulamayı çalıştırmak için Docker'a ihtiyaç duyulmaktadır. Kullanımı hatırlayın: "-p 8080:8080" WildFly'ın dinlediği portu bağlamak için kullanılır

Daha sonra tarayıcıdan <http://localhost:8080/base> adresini açabilirsiniz.



- İlk örneği hatırlayın

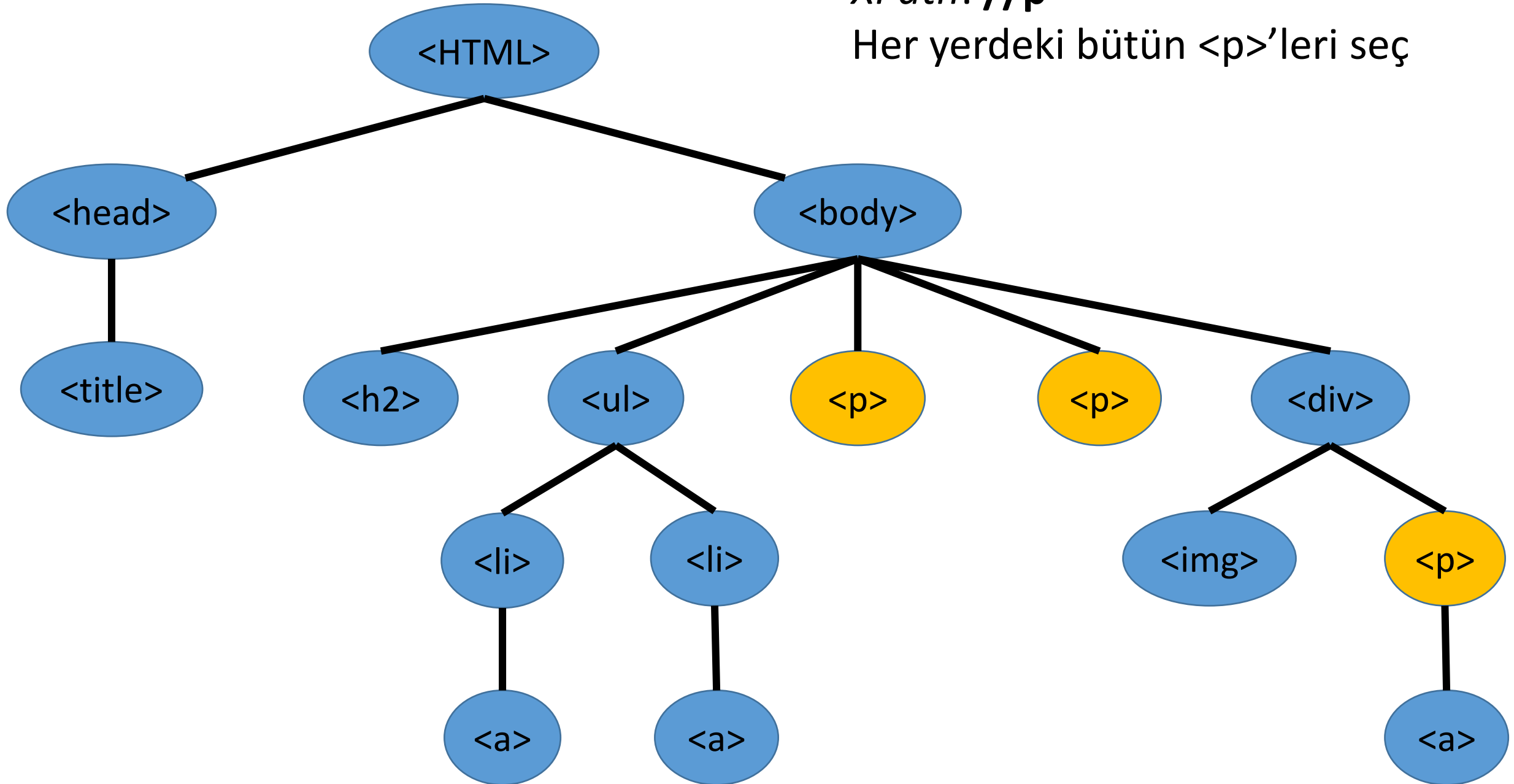


# XPath: Path İfadeleri

- “nodename”: verilen node adıyla her şeyi seç
- “/”: root node’unu seç
- “//”: geçerli alt ağacın içinde herhangi bir yeri seçin
- “.”: mevcut node’u seç
- “..”: mevcut node’un parent’ını seç
- “@”: özellik seç
- “\*”: bütün node’ları seç
- “@\*”: bütün özellikleri seç

*XPath: //p*

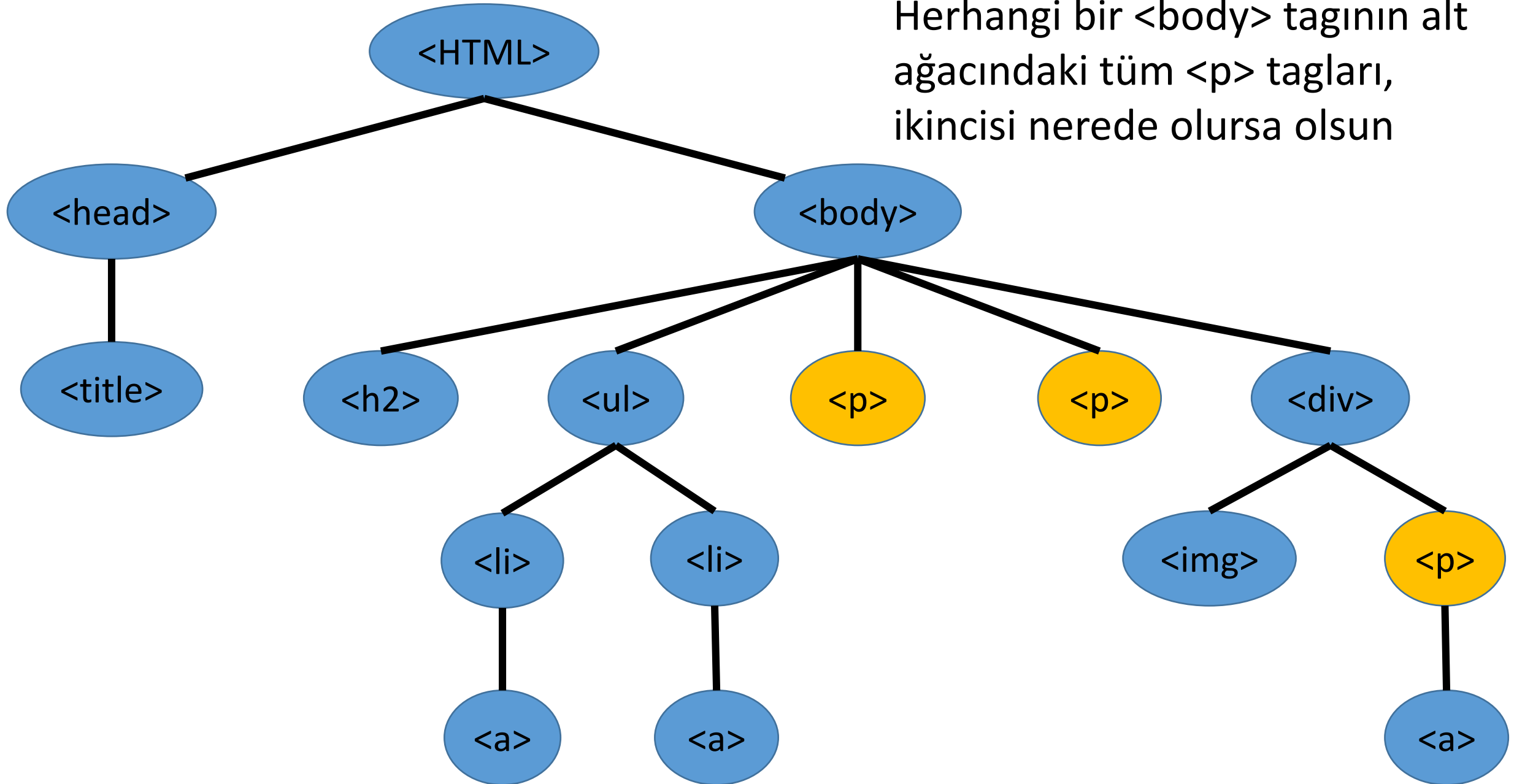
Her yerdeki bütün <p>'leri seç





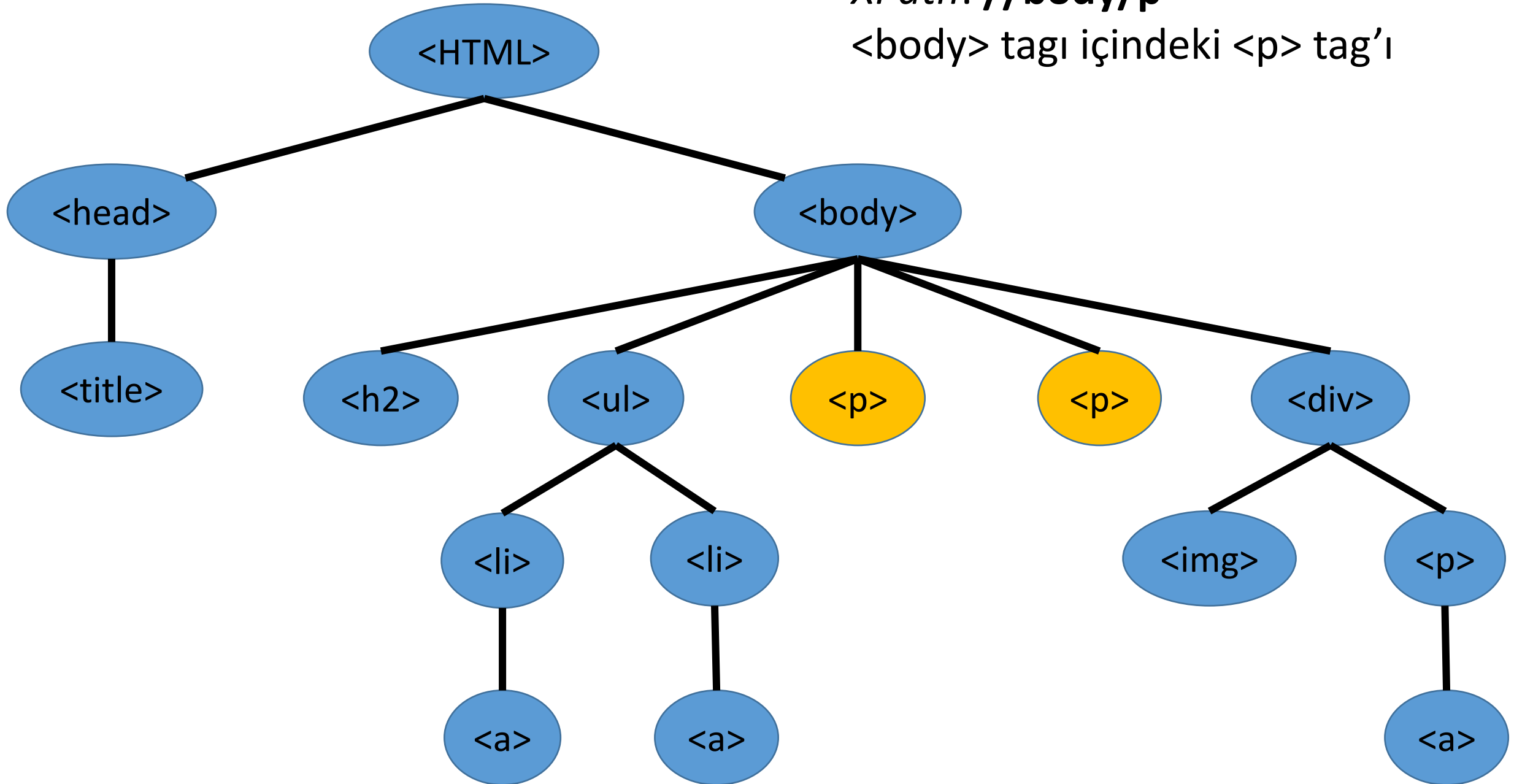
*XPath: //body//p*

Herhangi bir <body> tagının alt ağacındaki tüm <p> tagları, ikincisi nerede olursa olsun



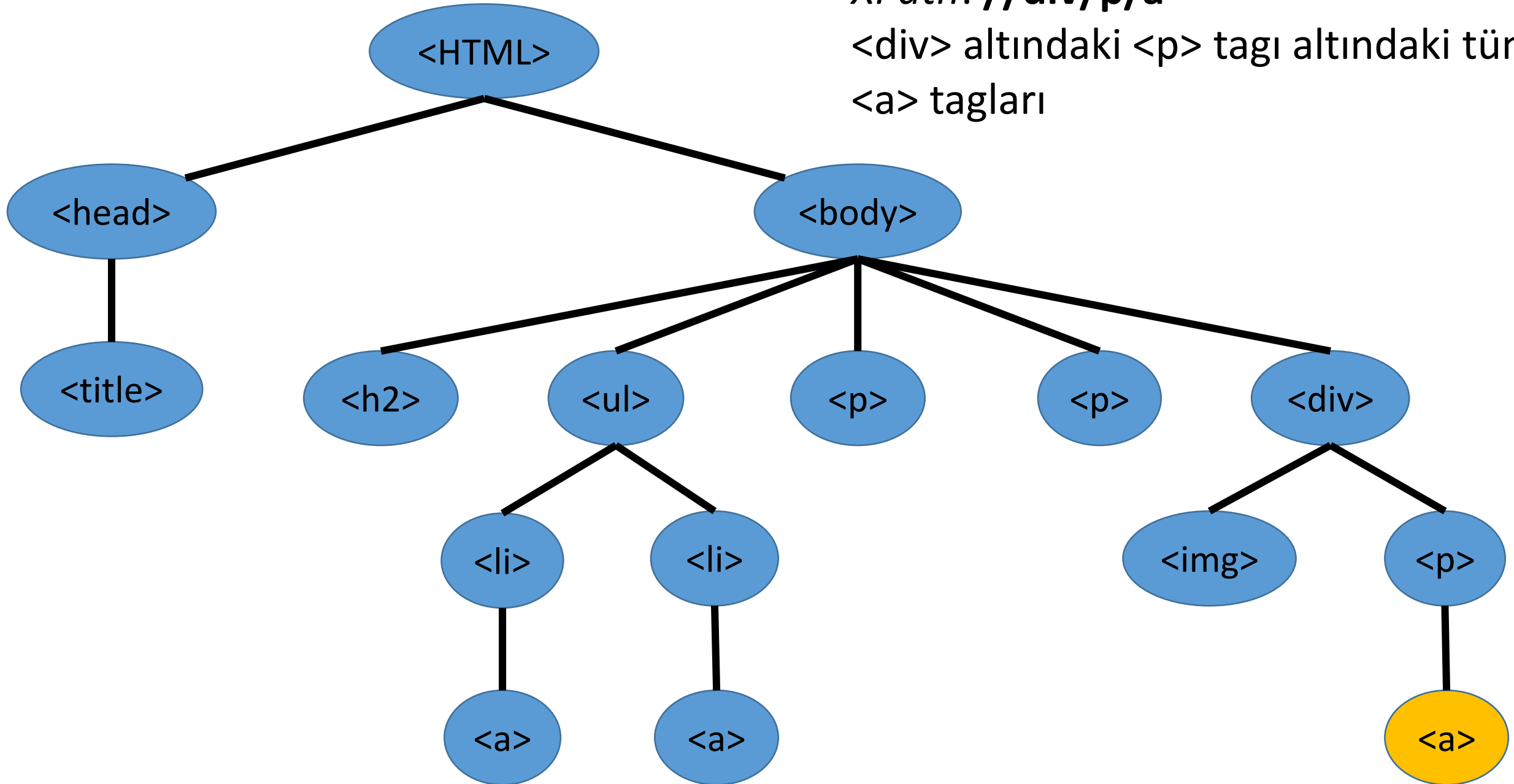
*XPath: //body/p*

<body> tagı içindeki <p> tag'ı



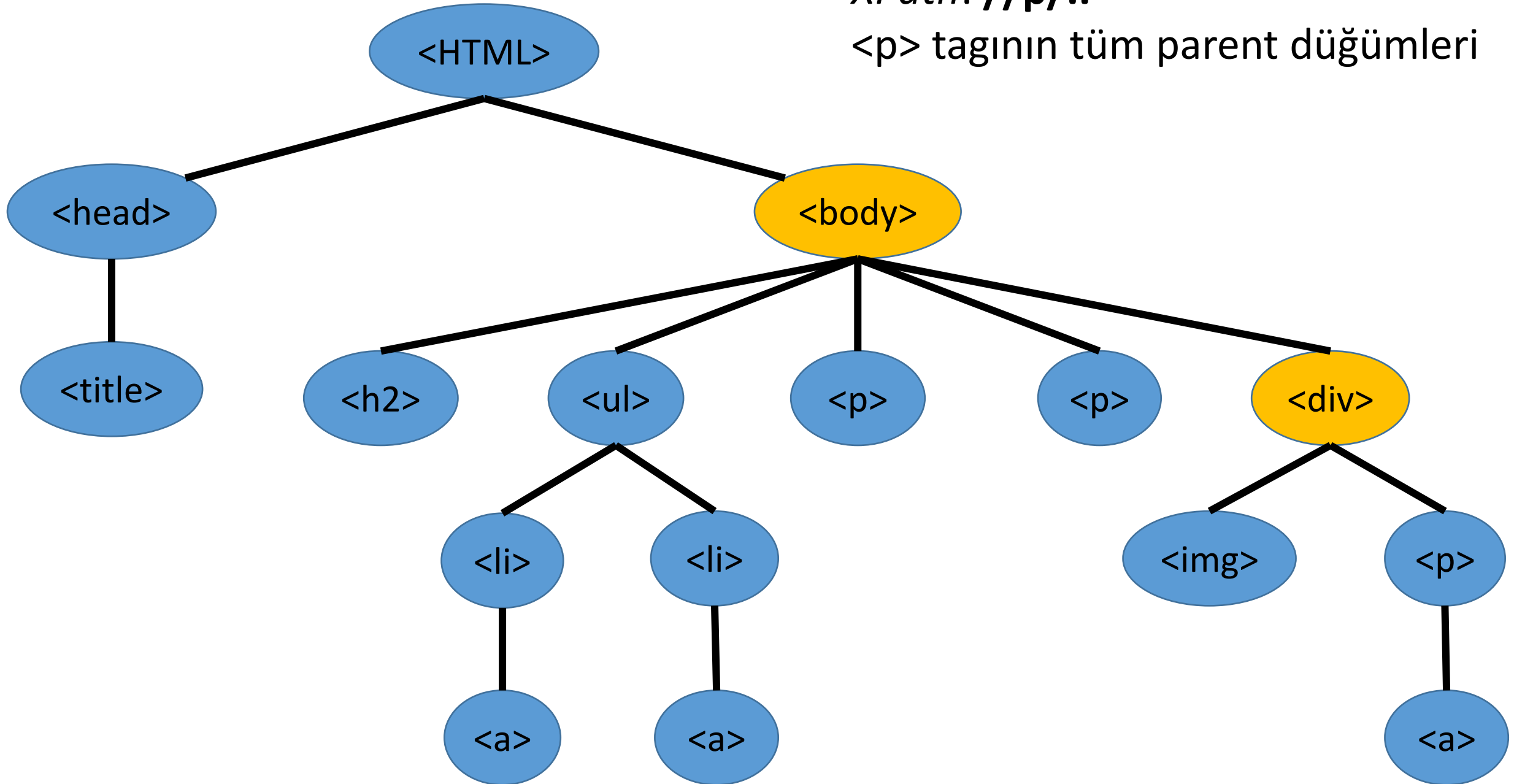
*XPath: //div/p/a*

<div> altındaki <p> tagı altındaki tüm  
<a> tagları



*XPath: //p/..*

<p> tagının tüm parent düğümleri

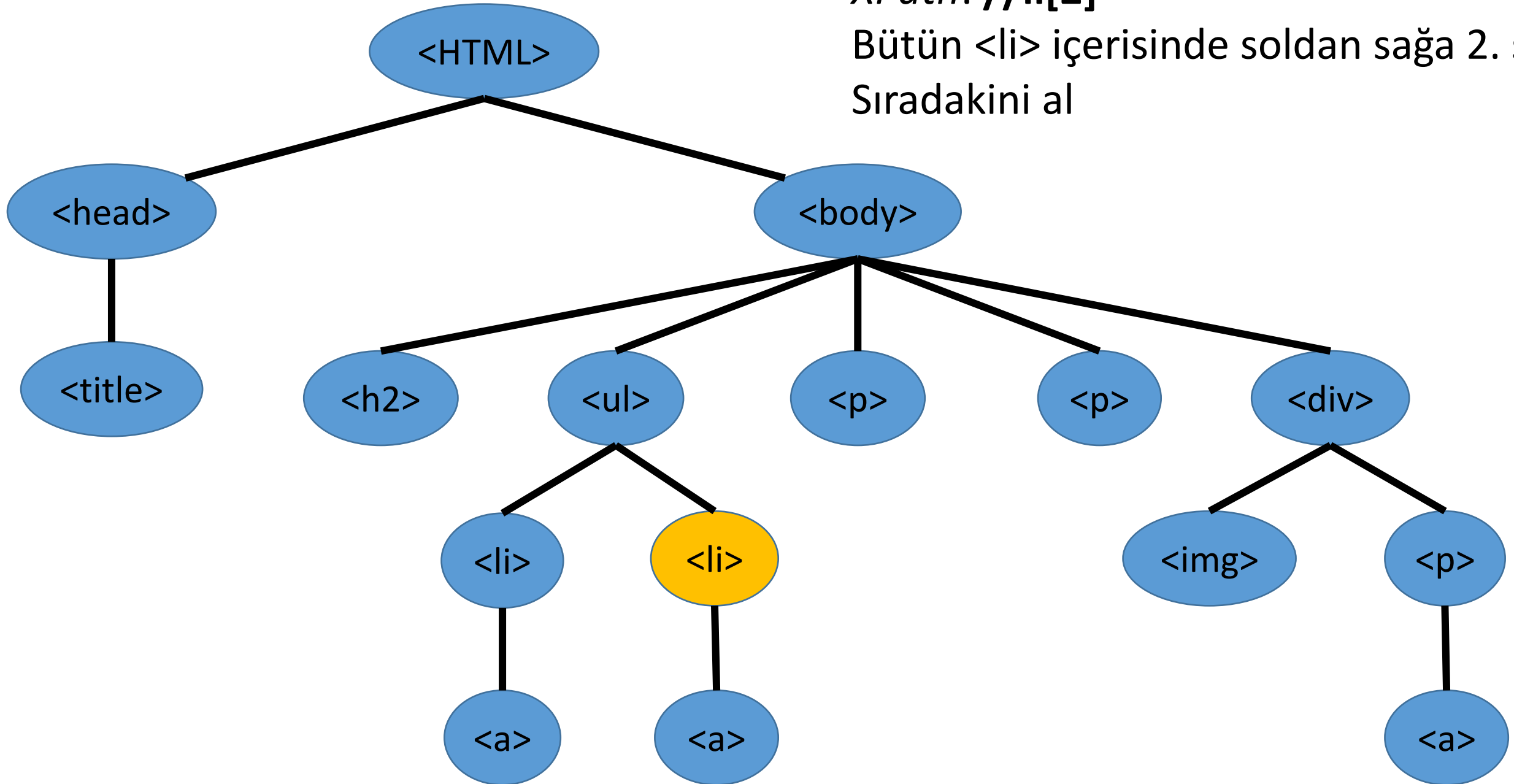


# XPath: Doğrulamalar

- Bütün eşleşen düğümler içerisinde «[]» içerisindeki eşleşen alt düğüm seti alınır
- Ayrıca XPath'den dönen elemanların indis değeri de olabilir
  - UYARI: 1'den başlar, 0'dan değil...

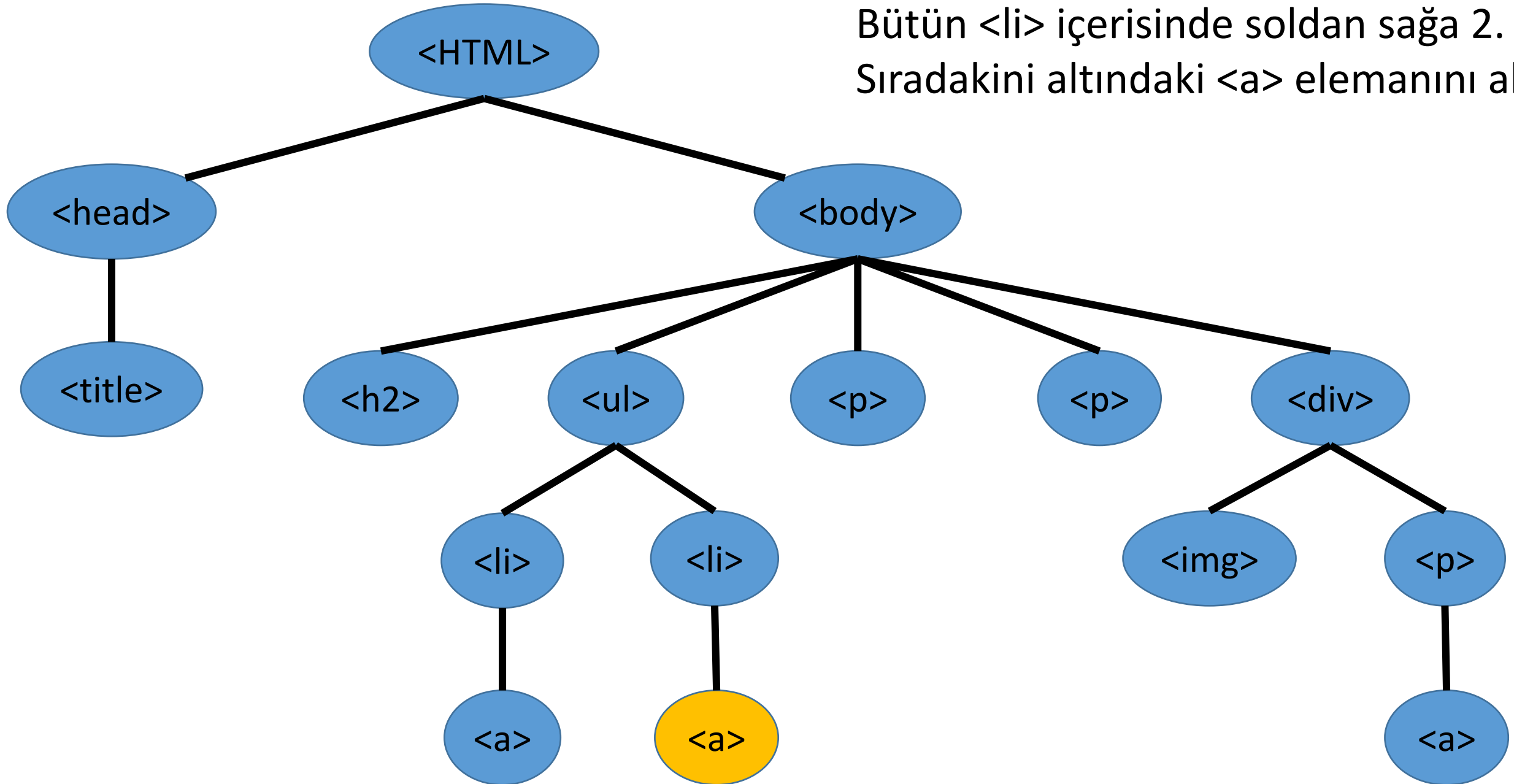
*XPath: //li[2]*

Bütün <li> içerisinde soldan sağa 2. s  
Sıradakini al



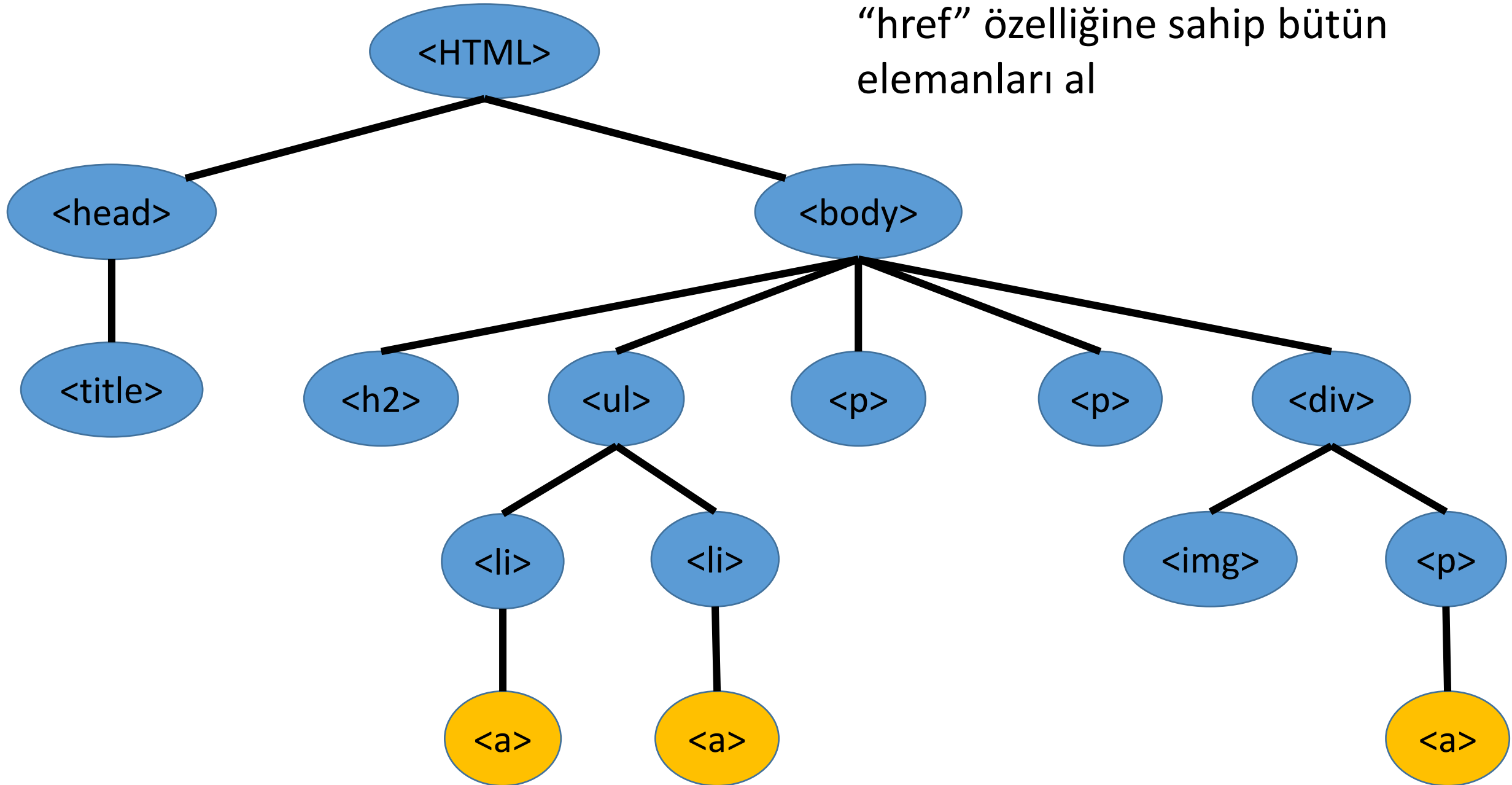
*XPath: //li[2]/a*

Bütün <li> içerisinde soldan sağa 2. s  
Sıradakini altındaki <a> elemanını al



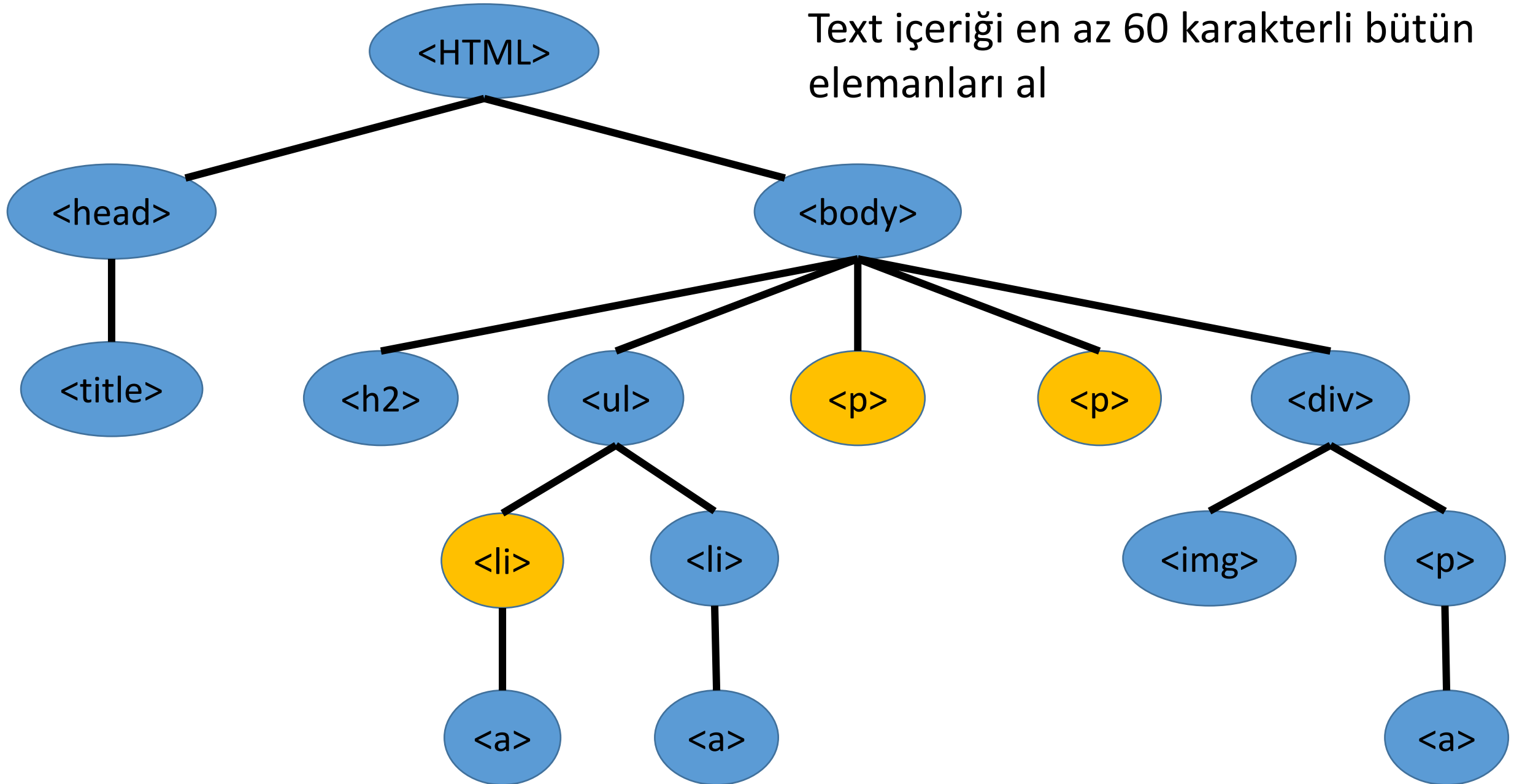
*XPath: //\*[**@href**]*

“href” özelliğine sahip bütün  
elemanları al





*XPath: //\*[string-length(text()) > 60]*  
Text içeriği en az 60 karakterli bütün  
elemanları al



- Chrome Developer Tool içindeki aramada XPath kullanılabilir
- Selenium testini çalıştırmadan XPath'in geçerliliğini kontrol etmek için kullanılması iyidir

Temel Örnek

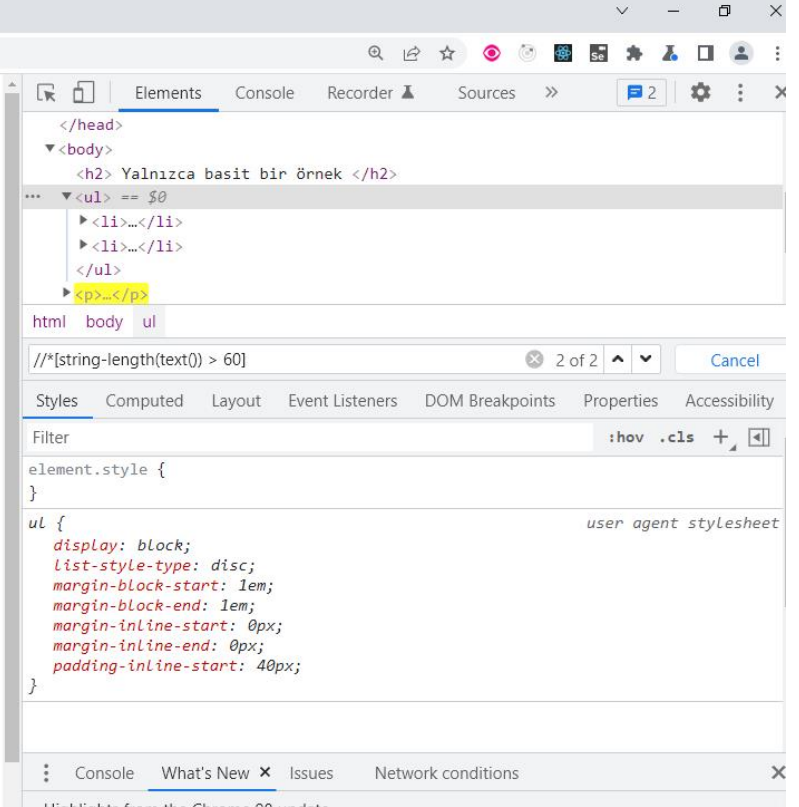

localhost:8080/base/home.html

### Yalnızca basit bir örnek

- webapp'teki farklı dosyalara bağlantı oluşturulabilir,örnek olarak: [foo/foo.html](#)
- Ancak WEB-INF altındaki dosyalar erişilemezdir, ör. [WEB-INF/web.xml](#) hata dönecektir

Bu uygulamayı çalıştırmak için Docker'a ihtiyaç duyulmaktadır. Kullanımı hatırlayın: "-p 8080:8080" WildFly'ın dinlediği portu bağlamak için kullanılır

Daha sonra tarayıcıdan <http://localhost:8080/base> adresini açabilirsiniz.



# Bakım

- Selenium testlerinde HTML öğelerini bulmak için XPath kullanmanız gerekir, örneğin butona tıklamak için
- XPath karmaşık olabilir
- Birçok farklı Selenium testinde aynı sayfayı geçmeniz gerekebilir.
  - ör, bir kullanıcı oluşturmak için Sign Up sayfası kullanımı
- Bir sayfadaki HTML değişebilir ve böyle bir sayfayı kullanarak her Selenium testinde XPath'i güncellemek zorunda kalmak istemezsiniz.

# Page Object (PO)

- PO, bir web sayfasında yapabileceğiniz tüm eylemleri içeren testte kullanılan bir Java sınıfıdır.
  - Her web sayfası için bir PO bulunur
  - Bir PO *“clickLoginButton()”* gibi metotlara sahip olabilir
  - Selenium testleri, HTML/DOM’a doğrudan erişmemeli, bunun yerine sadece PO'ları kullanmalıdır.
  - Bir PO'daki bir metot bir sayfa geçişini temsil ediyorsa (örneğin, bir bağlantıya tıklama), bu türde metot yeni sayfanın PO'sunu döndürmelidir.
  - Eğer HTML sayfası değişirse, yalnızca PO güncellenmelidir onu kullanan yüzlerce Selenium testi değil...
  - Ayrıca PO Selenium testlerini anlaşılması kolay hale getirir
-



Birim vs Sistem Testleri???





# Hangi Test Yazılmalı?

- Yalnızca birim test mi yazmalısınız?
  - Veya yalnızca Selenium testleri mi?
  - İkisinin karışımı mı?
  - Hangi test daha kullanışlıdır?
  - Hangisinin önceliği yüksektir?
  - Yeterli birim teste sahip olduğunuza karar verdiğinizde daha fazla Selenium testleri mi yazmalısınız?
-

# Birim Test (BT) Problemleri

- Çok fazla kod kolay (ör getters ve setter) kolay olabilir. BT bunlar için zaman kaybı olabilir
- Bir sınıfın diğer sınıflara karmaşık bağımlılıkları olduğunda, BT yazmak zahmetli olabilir
- Bütün sistemi BT seviyesinde test etmek için binlerce teste ihtiyaç duyabilirsiniz ve her bir test oldukça küçük bir kısmı test eder

# Sistem Testi (ST) Problemleri

- Genellikle BT'e göre **çok daha fazla** yavaştır
    - ör, tarayıcı başlat, sonra sunucu başlat, sonra butona tıkla ve veri gönder  
VS...
  - Eğer ST hataya düşerse sebebini bulmak çok daha zordur çünkü çok fazla kod çalıştırılır
-



# Yani, BT mi ST mi???

- Karmaşık olduğunu düşündüğün sınıflar/fonksiyonlar için BT yaz
  - Birim testleri debug etmesi kolaydır
- Bütün sistemin temel fonksiyonlarını ST ile test ettiğinden emin ol
  - Temel fonksiyonların çalıştığından emin ol bunlar pek çok temel kodu kapsayacaktır
- Eğer ST bug sebebiyle hataya düşerse BT yazıp debug ederek çözmeye çalış ve ilgili sınıfların entegrasyon testini tamamla (eğer bug'a sahipse düşündüğünüz kadar kolay olmayabilir)

# Test bir tür sanattır

- Ne yazık ki test bilimden çok sanata daha yakındır
- Genellikle hikayelere dayalı tecrübelerden oluşan pratik kurallar bütünüdür
- Bazı genel fikir birlikleri (ör., test önemlidir) olsa da bilimsel olarak bir rehber bulunmaz
- Örnek: tipik olarak literatürde daha hızlı çalıştırılabildiği için birim testlere daha fazla önem verilmesini önerir... *ancak çoğu uygulamada gerçekleşmez*
  - En azından enterprise seviyesinde uygulamalarda sistem ve entegrasyon testleri daha kullanışlıdır

# Mock Frameworkleri

- Birim testi yazmanın çeşitli zorlukları bulunmaktadır özellikle de harici bağımlılıklara sahip ise
- Örnek: veritabanı ile etkileşime geçen bir sınıfın birim testi nasıl yazılır?
  - Not: şimdiye kadarki görmüş olduğunuz veritabanı üzerindeki testler entegrasyon testiydi, çünkü bir veritabanı başlatıyor (H2 gibi) daha sonra container oluşturuyorduk (JEE/Spring)
- Bu durumun üstesinden gelebilmek için Mockito gibi çeşitli test frameworkleri bulunmaktadır. Bu frameworkler bağımlılıkları mock etmenizi sağlar
- Bu tür mock frameworkleri oldukça popülerdir ancak ben onları kullanmayı pek sevmiyorum (en azından enterprise sistem seviyesinde)
  - Testler sıklıkla kırılgan hale gelir ve bakımları zordur
  - Testler yanlış olabilecek dış bağımlılıkları test eder, yani gerçek şeyi test etmemektesiniz
  - Bu yüzden entegrasyon/sistem seviyesinde testleri yazmak daha iyidir

# Kod Kapsamı

# Kaç Adet Test?

- Kaç adet test yazılmalı?
- Yazabildiğiniz kadar yazmalı mısınız?
- Testlerin yeterli olduğu nasıl belirlenir?
- Gerçekten daha fazla test yazmanız gerektiğini ne zaman söyleyebilirsiniz?

# Kod Kapsamı

- Testler koşturulduğunda otomatik olarak kodun hangi kısmının çalıştırıldığı kontrol edilir
- Herhangi bir test tarafından çalıştırılmayan bir satırda bir hata olabilir ve çalıştırılmadığı için test de bunu tespit edemez.
- Çalıştırılan ifade yüzdesini hesaplayabilirsiniz
  - Ör: 200 satırın 160 satırı kapsanırsa 80% satır kapsamına erişilir
- Not: daha karmaşık kapsama kriterleri bulunmaktadır ancak ifade (statement) kapsamı en sık kullanılan/bilinenidir

# Kısıtlamalar

- Kod kapsamı iyi bir teste sahip olduğunuzu söylemez bunun yerine daha fazlasına ihtiyacınız olup olmadığını söyleyebilir
- %100 kod kapsamı? Yine de çok fazla bug bulunabilir
  - Bütün bug'lar uygulamanın çökmesi ile sonuçlanmaz
- %15 kod kapsamı? Test paketin gerçekten çok kötü, daha fazla test yazmaya devam et
- Genellikle %50-%80 aralığında kod kapsamına erişilmeye çalışılır
- 100% genellikle pratikte gerçekleşmez
  - Ölü kodlar, defansif programlama vs.

# Test Ekonomisi

- Müşteriler (yazılım) ürün satın alırlar ve test senaryoları ile ilgilenmezler
- Firmaların gelirleri yazılım ile gerçekleşirken testin de bir maliyeti vardır
- Bir çalışan her test yazdığında, müşterinin istediği yeni özellikleri yazmak için kullanacağı zamanı harcar.
- Ancak müşteriler de kusurlu ürün istemezler...
- Teste ne kadar çok zaman/kaynak yatırırsanız, kusur riski o kadar azalır
- Doğru denge, bug'a sahip olmanın ekonomik maliyetine bağlıdır.
  - ör, video oyunlarla banka uygulaması arasındaki farkı hatırlayın...





# Örnek: Öğrenci Projesi

- Verilen X zamanında daha iyi not almak için yeni özellik eklemek istemez misiniz?
  - Veya bunun yerine uygulamanın çökmesini engelleyen ancak notunuzu düşürecek olan testleri yazmayı mı tercih edersiniz?
    - ör, sınav sırasında bir demo açıyorsunuz ve kontrol eden hoca önünde uygulamanız çöküyor...
-

# Kapsam Ölçümü

- Java'da kapsam ölçümü için temel araç *JaCoCo'dur*
  - JaCoCo açılımı "Java Code Coverage"
- Maven plugin'i ile kolaylıkla aktif hale getirilebilir
- Testte herhangi bir değişiklik yapmanıza gerek yok
- Build sonunda bir rapor oluşturacaktır
  - ör, bir web sayfası
- Not: IntelliJ gibi IDE'lerin de dahili kapsam ölçen araçları bulunmaktadır

# report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Metl
 <a href="#">backend</a>	<div><div></div></div>	89%	<div><div></div></div>	100%	3	14	3	27	3	
 <a href="#">frontend</a>	<div><div></div></div>	89%		n/a	1	10	2	15	1	
total	14 of 136	89%	0 of 4	100%	4	24	5	42	4	

# Enstrümantasyon



JaCoCo gibi araçlar kapsamı nasıl ölçüyor?



Bir sınıf çalıştırıldığında, “.class” dosyasındaki bayt kod, Sınıf Yükleyiciler tarafından JVM'ye yüklenir.



Kod kapsamı araçları, bayt kodunun yüklenmesini engeller ve anında değiştirir



Bu değişikliklerle bir probe ekler, örneğin metot çağrısından sonra bir ifadenin çağırılıp çağırılmadığını görüntüleyen metot ekler

# Git Repository Modülü

- *NOT: açıklamaların büyük bir çoğunluğu kod içerisinde yorum satırı olarak bulunmaktadır, burada slaytlarda bulunmamaktadır*
- **intro/spring/testing/selenium/jsf-tests**
- **misc/test-utils**
- **intro/spring/testing/selenium/crawler**
- **intro/spring/testing/mocking**
- **intro/spring/testing/coverage/jacoco**
- **intro/spring/testing/coverage/instrumentation**
- Ders 09 alıştırmaları (dokümantasyona bakınız)