Projeto de Software 2019.1

Notas de Aula

Dalton Serey

Atualizado em 15/abril/2019

IMPORTANTE: Este material não foi planejado para servir de base de estudo. E, portanto, não é suficiente para estudar para a disciplina. Este material tem o propósito de me servir de orientação na apresentação do conteúdo nas aulas. Assim, use este material apenas como roteiro do conteúdo coberto, mas lembre-se que é fundamental **complementá-lo com leituras adicionais e exercícios**. À medida que eu for evoluindo no curso, atualizarei este documento. Peço que confira se há alguma versão mais atualizada no endereço abaixo.

Documento atualizado: http://www.dsc.ufcg.edu.br/~dalton/projsw/notas_de_aula.pdf
Link do curso: http://www.dsc.ufcg.edu.br/~dalton/projsw/

1. Como funciona a web?

> O que é uma conexão internet? O que são sockets? O que é um endereço socket? O que é uma conexão TCP? O que é HTTP? Exemplos com netcat/telnet/python. O que são e como são requests e responses?

Conteúdo abordado. Programas vs processos vs threads. Streams de entrada e saída. Sockets. Endereços IP, portas e endereços TCP/sockets. Conexões TCP X portas TCP. Características de conexões TCP. Uso de conexões TCP/sockets em Python. Netcat (cliente TCP universal). Dados binários vs textual. Bytes vs strings. Encodings. Protocolo textual. Princípios e ideologia da web. Características de HTTP: connectionless, stateless, textual, multi-nível. Origin-server, user agent, client, proxy. Requests e responses. Formato geral de mensagens HTTP: start line, headers, blank line, body (bytes!). Requests usam request lines. Conceitos de verbos/*métodos*, recurso e URI de HTTP. Responses usam status line. Parsing de mensagens HTTP. Status codes http: 1xx, 2xx, 3xx, 4xx e 5xx. Headers mais relevantes: Date, Connection, Host, Cache-Control, Accept, Content-Type (e mimetypes). Conceito e formato de mimetypes: `text/plain`, `text/html`, `application/json`. Sites estáticos X aplicações web. Aplicações clássicas X aplicações modernas.

aula 1: teórica | aula 2: prática: sockets + threads | aula 3: prática: servidor http | aula 4: teórica (encerramento do tema)

1.1. programas X processos X threads

- programa: texto que expressa os algoritmos/comportamento
- processo: abstração criada pelo SO: o programa em execução
    - cada processo tem (em geral):
        - uma linha de execução (ou thread)
        - sua própria “memória” (espaço de endereçamento virtual)
- ilusões criadas pela abstração do processo para o programador
    - que ele “está só” na máquina (o programador não precisa se preocupar com os demais programas que concorrem pela CPU)
        - na prática, o SO/HW tomam a CPU de cada processo periodicamente; a isso se denomina *preemção* (leia sobre multiprogramação — lembre que você voltará a todo este tema na disciplina de sistemas operacionais);
    - que tem acesso a toda a "memória" (eev) da máquina (logo, o programador não precisa se preocupar com problemas de concorrência)
        - na prática, o SO/HW mapeiam os dados do processo entre endereços virtuais (que o processo vê) e endereços reais na RAM (que só o SO/HW vêem);
- thread: abstração do SO para uma execução do código do processo

        - inclui a "linha do código" que está sendo executada
        - e uma **pilha de chamadas** de funções
- processos podem pedir ao SO para ter **mais threads**; assim
    - um único processo pode ter mais de uma linha de execução (thread) que estarão sendo executadas simultaneamente (sem importar se a máquina tem uma ou várias CPUs — multiprogramação ou multiprocessamento);
    - cada thread criada para um mesmo processo terá:
        - seu próprio registro da linha de código sendo executada
        - sua própria pilha de chamadas
    - mas, todas irão compartilhar a memória (ou EEV)

Em nossos exercícios práticos, iremos aprender os fundamentos de programação com threads em Python. Mas os conceitos se aplicam a toda linguagem de programação que use threads.

1.2. Streams, TCP, sockets e portas

- conceito e filosofia de streams (stdin, stdout, stderr)
    - bytes X chars X strings
    - encodings (encode X decode)
    - unicode X utf-8
    - exemplos com python2 X python3
    - exemplos de arquivos utf-8 X latin1
- comunicação interprocessos
    - exemplos com redirecionamentos e pipes
- endereços TCP:
    - endereços IP
    - portas
    - endereços socket
- exemplos com netcat/nc/telnet

Os exercícios de lab exploram sockets. São construídos pequenos servidores e clientes sockets. Combinando o tema com threads, é construído um pequeno servidor de chat que obriga o servidor a manter múltiplas conexões em threads.

1.3. Fundamentos do HTTP

- protocolo textual sobre sockets TCP, na porta padrão 80
- é a base da world wide web
- filosofia: protocolo de especificação aberta: todos podem adicionar um novo servidor/cliente e participar da web
- características fundamentais:
    - stateless, connectionless (no nível http)
    - baseado em ciclos independentes de requests e responses
- processos que participam têm diferentes papéis:
    - user agent ou client: emite requests;
    - origin-server: emite responses em resposta aos requests;
    - proxy: intermediário repassa requisições em nome do cliente (forward proxy) ou respostas em nome do servidor (reverse proxy); atua de forma bastante transparente;
- Sugestão de leitura: HTTP Made Really Easy

1.3.1 Mensagens HTTP

- http tem um formato geral para as mensagens
  - start line + headers + blank + body
  - toda linha é terminada com <CR><LF>

- requests
  - a start line é chamada de **request line**
  - sintaxe: <verbo> <recurso> <protocolo>
  - exemplo: "`GET /index.html HTTP/1.1`"
  - há um conjunto pequeno e pré-definido de verbos
  - o recurso é uma string que indica o objeto de interesse
  - o protocolo é meramente a indicação do protocolo usado

- responses
  - a start line é chamada de **status line**
  - sintaxe: <protocolo> <status code> <status text>
  - o protocolo indica o protocolo usado
  - o status code é um código numérico
  - o status text é uma versão textual/legível do status code

REQUEST: sintaxe

request line	<método/verbo> <path> <protocolo>
request headers 1 ou mais único obrigatório: HOST	<nome>: <valor> HOST: <host parte da url>
blank line	
message body opcional / payload	<corpo da mensagem>
Importante:	**Todas** as linhas terminam com <CR><LF>

Exemplo

```
GET /~dalton/teste
Host: dsc.ufcg.edu.br
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/37.0.2049.0 Safari/537.36
```

RESPONSE: sintaxe

status line	<status code> descrição do status
response headers são 0 ou mais	<nome>: <valor>
blank line	
message body opcional; representação do recurso	<body>
Importante:	**Todas** as linhas terminam com <CR><LF>

Responses incluem uma

Exemplo

```
HTTP/1.1 200 OK
Date: Wed, 06 Aug 2014 10:39:52 GMT
Server: Apache/2.2.11 (Unix) mod_ssl/2.2.11 OpenSSL/0.9.8e PHP/5.5.10
mod_perl/2.0.4 Perl/v5.8.8
Content-Location: teste.txt
Vary: negotiate,accept
TCN: choice
Last-Modified: Sat, 26 Jul 2014 04:14:47 GMT
ETag: "89895f-10-4ff10ee27fbc0;4ff11b3a05a00"
Accept-Ranges: bytes
Content-Length: 16
Content-Type: text/plain

Dalton. Em txt.
```

1.4 recursos

- os <recurso>s mencionados em request lines
    - são os objetos/dados armazenados por servidores (origin-server)
    - exemplos: textos, imagens, arquivos de dados, etc
- recursos, em http, são identificados por URIs
    - URI = universal resource identification
- um único recurso pode ter diversas *representações*:
    - em diferentes linguagens
    - em diferentes formatos (texto puro, http, pdf, doc, json, imagens, etc)
    - em diferentes níveis de detalhe, tamanho, resolução, etc
- um ciclo http trata exatamente do pedido e entrega de um recurso
- mais adiante voltaremos ao tema URI

1.5. Headers HTTP

- os headers são campos de meta dados que acompanham cada mensagem HTTP
- o header é formado por *fields* (propriedades) que são pares de nomes e valores
- os headers permitem adicionar detalhes a uma requisição
- é em headers que o cliente indica que características quer na representação recebida do recurso (linguagem, formato, etc)
- e o servidor usa os headers para indicar características da representação que, de fato, está sendo enviada (linguagem, formato, tamanho, etc)
- os headers também são usados para orientar o comportamento esperado dos intermediários (proxies)
- é através deles que um servidor pode pedir para que intermediários armazenem cópias de recursos muito utilizados, para evitar novas requisições

1.5.1 Headers importantes

- Alguns headers devem ser conhecidos por qualquer desenvolvedor web
- Date, Connection, Host, Cache-Control
- À medida que evoluírmos no curso, veremos outros
- Dois deles, contudo, merecem destaque desde já: Accept, Content-Type

- Um conceito importante antes de prosseguirmos é o de MIME type
    - é, essencialmente, um padrão que permite expressar a natureza e o formato específico de um recurso, documento ou arquivo qualquer
    - pense nele como a forma mais genérica da conhecida "extensão de arquivos"
- Um MIME type tem a forma "`<tipo>/<subtipo>; <parâmetros>`", onde
- Exemplos que mais usaremos:
    - `text/plain`: para texto puro (codificado em **utf-8**)
    - `text/html`: para texto com marcação html (**us-ascii**)
    - `application/json`: para indicar dados em json (**utf-8**)

1.6. Sites estáticos X Aplicações Web

1. Web 1.0 se iniciou como uma coleção de sites estáticos
    a. o método GET era praticamente o único usado
    b. cada servidor tinha um conjunto estático de páginas para servir
    c. ver www.dsc.ufcg.edu.br como exemplo (Apache)
    d. forma primitiva de aplicação web: páginas estáticas + formulários

2. Aplicações web clássicas
    a. páginas HTML geradas pelo servidor
    b. cada requisição ⇒ uma consulta ao BD + geração de HTML
    c. páginas com formulários para coletar mais dados
    d. links para mudanças de tela (uma tela ~ uma página)
    e. frameworks escondiam HTTP: metáfora de objetos distribuídos + RPC/RMI

3. Aplicações web modernas
    a. app se apoiam transparentemente em HTTP
    b. servidores provêem dados em json/xml (mimetype `application/json`)
    c. uso extensivo de AJAX

2. HTML + CSS + JavaScript

2.1. O que é HTML?

1. HTML = **Hypertext Markup Language**

2. **Hypertext** = texto em que páginas são nós de um grafo; html permite conectar páginas de diferentes autores/fontes facilmente;

3. **Markup** = anotação: a ideia é que você “anota” o conteúdo para que o browser saiba como manipular esse conteúdo; html é legível por humanos, não apenas para a máquina;

4. **Language** = há **sintaxe** e **semântica** bem definidas! Logo, há html bem formado e há html mal formado; exemplos com: closing tags, etc;

2.2. O que é CSS?

5. CSS = **Cascading Style Sheets**

6. **Cascading** = cascading se refere ao fato de que a especificação prevê a possibilidade de combinar múltiplas folhas de estilo para a renderização de um documento;

7. **Style Sheets** = folhas de estilo é o termo usado em diagramação visual para se referir ao conjunto de regras que determinam os aspectos visuais, gráficos e de diagramação dos elementos de um documento ao se produzir sua visualização;

2.3. O que é JavaScript?

8. Linguagem de programação ⇒ permite adicionar **comportamento e aspectos dinâmicos** a um documento

9. Permite que o usuário interaja com a renderização do documento e, indiretamente, com o servidor HTTP

10. Em última instância**,** é o componente que permite **tornar meras páginas na internet em aplicações**.

HTML x CSS x JavaScript

1. HTML = conteúdo e estrutura (lógica)
2. CSS = estilo, aparência e disposição (layout)
3. JavaScript = comportamento, dinâmica, lógica

Parada para construir um exemplo completo com HTML, CSS e JavaScript. Uma página simples com conteúdo simples, um pouco de estilo e o mínimo de comportamento. Foco na separação de responsabilidades entre HTML, CSS e JavaScript.

Exemplo criado em sala de aula.

HTML x CSS x JavaScript

4. HTML = conteúdo e estrutura (lógica)

5. CSS = estilo, aparência e disposição (layout)

6. JavaScript = comportamento, dinâmica, lógica

Relembrando os aspectos centrais desta lição: 1) HTML é conteúdo e estrutura, CSS é estilo, aparência e disposição, e JavaScript é comportamento; 2) HTML e CSS são linguagens e têm sintaxe e semântica bem definidas.

3. HTML

3.1 Sintaxe de HTML

Um documento HTML é descrito através de um código fonte que é um simples arquivo de texto, contendo, essencialmente, tags e trechos de texto. Se o html é armazenado em arquivo, tipicamente usaremos extensão html. Se o html é produzido dinamicamente, o mime type correspondente é `text/html`. Seguem os principais conceitos:

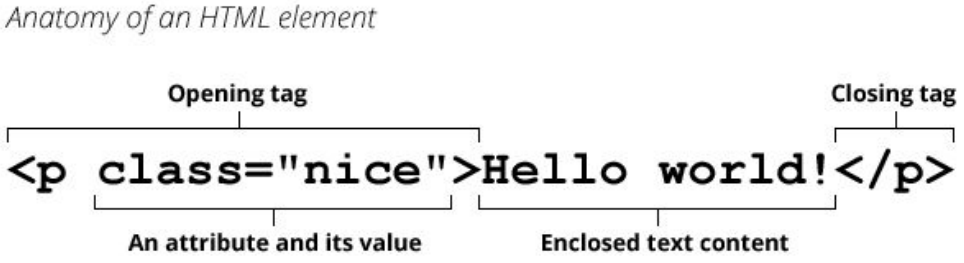
1. tags: **são trechos de texto de páginas HTML delimitados por *angle brackets*** *(* "<" e ">"), tipicamente usados para especificar elementos (ver abaixo) de um documento HTML; dentro dos *brackets* se coloca o nome do elemento e, opcionalmente, uma sequência de atributos e valores; importante: não é permitido espaço entre o angle bracket e o nome da tag; mas são obrigatórios espaços entre o nome da tag e o primeiro atributo (ver adiante) e entre os atributos;

    a. OBS: um registro pra quem ainda usa XHTML: não há mais self-closing tags!

    b. OBS 2: para os que gostam da história da tecnologia, sugiro darem uma olhada neste excelente quadro da história das tags html

2. elements: um *element* é simplesmente um item ou uma parte de um documento HTML; os elementos são descritos, no código fonte da página, através de *tags*; alguns elementos requerem um par: tag de abertura (opening tag) + tag de fechamento (closing tag); outros elementos são descritos apenas por uma única tag (nesse caso, a tag de fechamento não é apenas desnecessária, é proibida pelo padrão atual);

3. **content**: o conteúdo em si do elemento; em termos do código fonte, se refere a tudo que estiver entre as tags de abertura e de fechamento de um *element*



qualquer;

4. **attribute**: é uma característica ou propriedade modificável de um elemento; no código fonte, atributos são colocados **na tag de abertura** em que se definem pares nome, valor, separados por um "=";
    a. é boa prática sempre colocar aspas duplas ao redor do valor, mas não é obrigatório se o valor não contiver espaços;
    b. também se pode usar aspas simples; não se costuma usar espaços ao redor de `=`, mas sempre deve haver espaço antes do nome de cada atributo;

Alguns extras sobre html/tags/elementos...

5. **id** é um atributo de especial importância; todo elemento pode ter um id; seu valor deve identificar um elemento de forma única (se isso não ocorre, é um erro na formação do html e pode levar a erros); é muito usado para programar e acessar os elementos mais facilmente; ver com exemplo; os browsers dão acesso direto a todo elemento com id, definindo um nome (variável) de escopo global igual ao id e que dá acesso ao elemento;

6. espaços adicionais são, em geral, ignorados pelo interpretador html;

7. **doctype não é uma tag**: é apenas uma diretiva para o browser saber como interpretar devidamente o restante do conteúdo; todo html5 deve ser iniciado um doctype apropriado; com ele nos asseguramos que o browser usará o engine apropriado e também que o html segue as convenções, sintaxe e semântica de html5;

8. **head**: é uma tag que descreve reúne declarações gerais da página mas que não fazem parte do documento em si; permite, por exemplo, indicar o conjunto de caracteres e encoding usado; também permite indicar arquivos externos que devem ser lidos para, como as folhas de estilo, por exemplo;

9. **meta**: usaremos para determinar a tabela de caracteres (não tem closing)

10. **title**: o título (obrigatório)

11. **body**: é onde o conteúdo e a estrutura em si da página vão; é o que contém o conteúdo visível para o usuário;

3.2. Semântica de HTML

HTML é uma linguagem. Como tal, tem sintaxe e semântica definida. A semântica de HTML, contudo, determina como o código fonte com todas as tags e textos deve ser interpretada e renderizada (apresentada visualmente) pelo browser para o usuário.

Concretamente, a semântica de HTML determina como o código-fonte será transformado em uma estrutura de dados “viva”, chamada DOM (document object model).

Ao DOM são integradas as propriedades definidas nas folhas de estilo CSS. É a partir dessa estrutura de dados resultante que o módulo de renderização produz a visualização do documento apresentada ao usuário.

3.3. História e recursos para trabalhar em HTML

1. Até 1997: cada browser decidia o que colocar em sua versão de HTML
2. Em 1997: W3C cria HTML4.0 e HTML4.01
3. Em 2000: XHTML
4. Em 2004: WHATWG é criado
5. Em 2007: WHATWG se junta a W3C
6. Em 2011: HTML5
7. Como funciona agora? W3C faz standards; WHATWG mantém HTML vivo (living standard); muda todo dia continuamente; aos poucos o que realmente funciona bem, vira standard pelo W3C;
8. É nosso trabalho, como desenvolvedores, estar à par da evolução e do atual estágio desses standards. Em particular, porque os principais browsers são continua e silenciosamente atualizados (ever green). Logo, todo dia há mudanças no html, css e javascript disponíveis no ambiente.

9. Então, onde achamos informações sobre HTML, CSS e JavaScript?
    a. https://www.w3.org/TR/html5
    b. https://html.spec.whatwg.org
    c. http://caniuse.com
    d. https://validator.w3.org

3.4. Modelos de Conteúdo HTML

Lição 1. Modelos de conteúdo de HTML4

1. O que são modelos de conteúdo?
    a. regras que definem o que cada elemento pode conter internamente
    b. e como o browser vai interpretar e renderizar o elemento e seu conteúdo
    c. OBS: css pode mudar a renderização!

2. Em HTML4 há 2 modelos de conteúdo:
    a. elementos **block level** (div é a tag mais genérica do tipo)
    b. e elementos **inline** (span é a tag mais genérica deste tipo)
    c. OBS: mais adiante usaremos bastante div e span, para construirmos componentes;

3. Características de elementos *block level*:
    a. são renderizados em uma nova linha
    b. podem conter elementos de qualquer tipo internamente
    c. ocupam a largura completa disponível para uma linha de conteúdo

    d. exemplos: h1, h2, …, p, table, div, …

4. Características de elementos *inline*:
    a. são renderizados na mesma linha atual
    b. só podem conter elementos inline internamente (não block level)
    c. ocupam o mínimo de largura possível
    d. exemplos: em, code, a, img, input, button, span, ...

3.5. Modelos de conteúdo de HTML5

1. Em HTML5 passamos para 8 modelos de conteúdo!
2. Contudo, felizmente, ainda é possível relacioná-los ao modelo antigo
3. Eis os 8 modelos:
    a. Flow (veja este tipo como “equivalente” a block level)
    b. Phrasing (veja este modelo como “equivalente” a *inline*)
    c. Heading
    d. Sectioning
    e. Interactive
    f. Embedded
    g. Metadata

3.6. HTML5 (*semantic HTML*)

1. Semântica = significado: a ideia é que os elementos carreguem informação sobre o papel que o elemento desempenhará no texto/documento e que a informação de aparência seja secundária (em princípio, inexistente).
2. Observe que a "*semântica*" a que nos referimos aqui transcende a semântica no sentido de interpretação do html pelo browser e a respectiva construção do DOM; ela envolve a compreensão humana dos elementos de texto usados.
3. Vejamos os principais elementos semânticos de html5
    a. sectioning
        i. header, main e footer
        ii. nav e aside
        iii. section e article
        iv. address, time, figure, figcaption
        v. além dos já conhecidos: h1, h2, h3, h4, h5, h6 e hgroup

4. O uso apropriado de semantic markup:
    a. reduz o tamanho do código html
    b. reduz o tempo de processamento do html
    c. permite que bots, crawlers, readers sejam muito mais eficientes

d. facilita a leitura do seu código, porque inclui intenção dos elementos

5. Observações importantes
    a. **main só deve ser usado uma vez**: por especificação, apenas um elemento deve ser main em toda a página;
    b. **footer e header não precisam ser únicos**: ao contrário de main, é perfeitamente possível e aceitável que mais de um header/footer exista na página; **contudo**, devem se restringir a um para cada seccionamento da página (section content: article, section, aside)
    c. **article** indica um elemento totalmente “destacável” e “republicável“ (em outro site/página) do conteúdo
    d. **section** indica o que o próprio nome diz: uma seção; algo que é parte de um todo; um section é algo que é perceptível ao leitor humano como parte do todo; algo que apareceria em índices e/ou listas de conteúdos; não confunda com um div que é meramente uma parte estrutural de um componente no DOM, mas que provavelmente não será relevante para o leitor humano a ponto de merecer uma indicação no índice ou listagem de conteúdo; o div é meramente decomposição para fins de programação e/ou estilização;
    e. sections podem ser parte de articles: não há restrições aqui;
    f. articles podem ser parte de sections: também não há restrições aqui, mas use com cuidado; imagine uma página pessoal dividida nas seções:

apresentação, currículo, blog e artigos; certamente, essas partes são **section**s da página principal que merecem ser indexadas (colocadas em um índice ou listagem de conteúdo para apresentar ao usuário); e ainda assim, faz sentido que cada um dos posts do blog sejam tratados como **article**s, para que possam ser automaticamente destacadas da página principal e reusadas em outros contextos;

g. **aplicações também se beneficiam de markup semântico**: alguns pensam que apenas páginas de texto se beneficiam de markup semântico, mas isso não é verdade; aplicações também se beneficiam; praticamente todas as aplicações têm áreas que se podem marcar como header (para informações gerais do site/página ou do usuário identificado, por exemplo), footer (para informações de contato, data de atualização, versionamento, etc), main (para o conteúdo principal)

3.6.1 Leituras recomendadas sobre semantic html

- Semantic HTML
- Stop using so many divs!

4. CSS

1. Conteúdo é importante, mas… a **aparência** conta muito…
    a. a absorção de conteúdo pelo usuário depende de sua aparência
    b. usuários tendem a ficar mais tempo em sites mais agradáveis
    c. usuários tendem a retornar mais vezes para sites agradáveis
    d. aparência aqui inclui todos os aspectos visuais com que a página precisa ser renderizada: fontes, cores, tamanhos, posicionamentos, margens, etc...

2. CSS = Cascading Style Sheets
    a. é uma linguagem para descrever como renderizar conteúdo HTML
    b. permite renderizar de formas **muito** variadas um mesmo conteúdo
        i. visite CSS Zen Garden a título de exemplo e inspiração

3. CSS é o domínio que devia ser tipicamente de um designer…
    a. … mas, desenvolvedores precisam conhecer a tecnologia
    b. em particular, não é raro que devs precisem “implementar” designs: ou seja, ainda é comum que o programador precise produzir css para implementar um conjunto dado de especificações visuais (vindas de um designer)

4.1. Regras CSS

1. Sintaxe de uma regra
    a. seletor + bloco de declarações
    b. Exemplo: `p { color: red; font-size: 14px; }`
    c. seletor = especificação que diz a quais elementos a regra se aplica
    d. bloco de declarações = especifica as propriedades e valores dos elementos
2. Propriedades são pré-definidas pela especificação de CSS
3. Valores são também pré-definidos e para cada propriedade há um conjunto pré-definido de valores pré-definidos que podem ser usados, isso inclui
    a. vários tipos de valores enumerados
    b. valores numéricos
4. Toda propriedade é separada do valor por "`:`"
5. Todo valor é terminado por um "`;`" (opcional, mas é péssima prática não incluí-la)
6. Espaços são ignorados ao estilo de HTML, mas use indentação
7. Uma coleção de regras é chamada de uma folha de estilos: stylesheet

4.2. Selectors e o DOM

1. DOM = Document Object Model
    a. é essencialmente uma árvore (E.D.) na memória que representa o que é mostrado ao usuário pelo browser
    b. os nós da árvore são os elementos do HTML; se um elemento contém outros elementos dentro dele, tais elementos serão representados como nós da respectiva sub-árvore (ver exemplos no browser)
    c. alterar a árvore é a forma de alterar o que deve ser mostrado ao usuário; sempre que uma alteração for feita em qualquer elemento da árvore, o engine de renderização irá automaticamente atualizar a tela do usuário para refletir as mudanças feitas
    d. o DOM disponibiliza uma API JavaScript para navegar e manipular a árvore (mas veremos isso mais adiante)
    e. o browser explora a API do DOM para navegar pela árvore fazendo as modificações que forem especificadas por cada regra CSS
    f. ATENÇÃO: usaremos a nomenclatura convencional de estruturas de dados para nos referirimos aos elementos do DOM em relação a outros elementos do DOM; assim, um elemento é um nó; pai, filhos, descendentes, ascendentes, irmãos, etc se referem a nó(s) em relação a outro(s) no sentido convencional usado em árvores tanto em teoria dos grafos, quanto em EDs;

às vezes nos referiremos à hierarquia (ou escopo) abaixo de um nó, para nos referirmos aos elementos na subárvore cuja raiz é o nó dado;

2. Seletores CSS especificam os elementos do DOM afetados pelas declarações
    a. especificar os seletores corretamente é habilidade fundamental para controlar a visualização de uma página ou aplicação web
    b. vários frameworks e bibliotecas usam a mesma API de seletores do browser para associar comportamento ou até para manipular o DOM

3. Seletores de elementos, classes e de IDs
    a. nomes de elementos como seletores: `p {...}`
    b. classes como seletores: `.aviso {...}`
    c. IDs como seletores: `#usuario {...}`

4. Já discutimos sobre os elementos html… mas e classes e IDs?
    a. classes são especificadas pelo atributo class em cada elemento
    b. qualquer tipo de elemento pode pertencer a uma classe
    c. vários elementos podem pertencer à mesma classe
    d. já IDs são especificados pelo atributo id em cada elemento
    e. apenas um elemento pode ter um dado id em todo o html

5. Seletores podem ser agrupados, separando-os por vírgulas
6. Combinators: seletores também podem ser combinados, eis os mais importantes
    a. Elemento de certa classe: `p.aviso {...}` (cuidado, SEM espaço!!!)
    b. Seletor de filho (imediato/direto): `footer > p {...}`
    c. Seletor de descendente: `footer  p {...}` (cuidado, COM espaço!!!)

7. Usando seletores em JavaScript para selecionar elementos no DOM
    a. jQuey: biblioteca para navegar e selecionar elementos do DOM que explora o sistema de seletores de CSS (2006)
    b. pouco depois, foi especificada a Selectors API, em que foram definidos diversos métodos de acesso ao DOM, ao estilo de jQuery; entre eles querySelector() e querySelectorAll()
    c. a função querySelectorAll(seletor) retorna um array (na verdade, um NodeList) contendo todos os elementos que atendem à especificação dada pelo seletor
    d. já a função querySelector(seletor) retorna o primeiro elemento encontrado pelo algoritmo de encaminhamento usado (que é um encaminhamento de pré-ordem em profundidade)
    e. Links de interesse:
        i. a especificação em: https://www.w3.org/TR/selectors-api2/
        ii. localizando elementos usando a API de Seletores

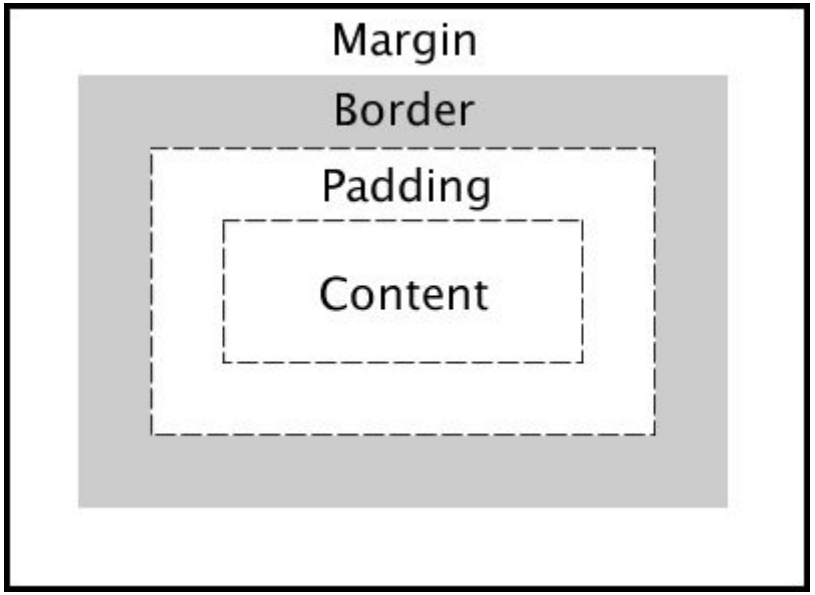
8. Pseudo classes e pseudo-elementos
    a. Pseudo-classes permite selecionar elementos em certo *estado*
        i. São exemplos: hover, disabled, focus, first-child, first-of-type…
        ii. Sintaxe: a:hover, a:visited, p:hover
        iii. Mais sobre pseudo-classes
    b. Pseudo-elementos permitem selecionar *partes* de elementos
        i. São exemplos: first-letter, first-line, before, after
        ii. Sintaxe: a::first-letter, p::first-line
        iii. Mais sobre pseudo-elementos
    c. Parar para mostrar exemplos reais: menus com links

4.3. Algoritmo de cascateamento (*cascading*)

1. Lógica pelas quais o browser combina regras de diferentes origens
2. Determina os valores de properties que devem ser aplicados quando…
    a. … um elemento está coberto por diversas regras
    b. … e até quando as regras parecem conflitantes
3. As regras são:
    a. Declarações são fundidas (*merged*) se não são conflitantes
    d. Declarações são herdadas pela estrutura do DOM
    e. Última declaração ganha (considerando a ordem de processamento)
    f. Declarações com maior especificidade ganham (esta regra é uma das maiores fontes de confusão!); veja ordem de especificidade de seletores:
        i. inline styling (mais alta especificidade)
        ii. style por ID
        iii. classes, pseudo-classes e atributos
        iv. elementos e pseudo-elementos (mais baixa especificidade)
4. A exceção `!important`
    a. regra geral: NÃO USE se todo o css é seu apenas
    b. a exceção: em casos específicos, quando usar CSS de terceiros

4.4. Renderização e o *box model*

1. Todo elemento é tratado como uma caixa (box)
2. Cada caixa não contém apenas o *conteúdo*; ela inclui:
    a. contents (o conteúdo propriamente dito; nível mais interno do box; visível)
    b. padding (área deixada ao redor do conteúdo; é transparente)
    c. border (área que fica ao redor do *padding*; pode ser visível)
    d. margin (área deixada ao redor da borda; é transparente)



3. Tutorial sobre o box model

4.5. Layout (tables, floats, position, flexbox e grids)

1. Layout em css é um dos temas mais cheio de armadilhas
2. Historicamente, já se fez layout web...
    a. … com tables (obriga a colocar o layout no html)
    b. … com divs e position
    c. … com divs e floats (reduz tags no html, mas é uma gambiarra)
3. Mais recentemente, devemos preferir
    a. … flexbox, para layouts lineares (unidimensionais)
    b. … grids, para layouts bidimensionais
4. flexbox
    a. ideia: sequências de elementos (semelhante a inlines)
    b. conceitos: em flex há containers e items
    c. flex container: use "display: flex" para identificar
    d. flex item: elementos filhos imediatos de flex containers são items flex
    e. properties mais importantes: flex-direction, flex-wrap, justify-content, align-content, align-items, order, flex-grow, flex-shrink, etc
    f. permite fazer layouts completos (é muito usado hoje), combinando múltiplos níveis de containers flexbox
    g. tutorial flexbox, outro tutorial

5. Grids
    a. ideia: semelhante a flexbox, mas bidimensional
    b. conceitos: containers e items
        i. o container define um grid
        ii. items são dispostos em células/areas do grid
    c. grid container: use "display: grid"
    d. grid items: elementos filhos imediatos são items
    e. properties mais importantes: grid-template-columns, grid-template-rows, grid-row, grid-column,
    f. item de grid podem conter outros containers grid e/ou flexbox
    g. permite fazer praticamente qualquer layout de forma perfeita, com uma especificação simples registrada apenas no CSS e não no HTML
    h. tutorial grid

Um web app completo (frontend only)

Aula Demonstrativa.Demonstração de um minúsculo app, incluindo html, css e javascript necessários. O app deve ser totalmente frontend, sem requerer qualquer dado de backend. Algo realmente trivial do ponto de vista funcional e de requisitos. A ideia é permitir que o aluno perceba o *big picture* e seja capaz de ver todos os componentes e tecnologias envolvidos. O aluno deve perceber que: 1) três arquivos fazem um webapp frontend (html, css, js); 2) que o backend aqui é mero host e provedor com alcance global dos arquivos e do app; 3) que HTTP é o protocolo usado e que requer ao menos 3 requests (um pro html, outro pro css e mais um pro javascript); 4) que cabe ao browser fazer a interpretação dos 3 arquivos e que HTML => DOM, CSS => visualização e JS => comportamento; 4) que diferente de programas simples de linha de comando, a execução do app web corresponde a uma inicialização do app e que, depois disso, ele permanece em memória até que a aba em que é acessado seja fechada; 5) que o app consiste em um conjunto de funções que devem ser executadas em reação a eventos; 6) que os eventos são tipicamente produzidos pelo usuário, mas que podem ser produzidos pelo próprio app ou pela "web"; 7) que a *entrada* corresponde a eventos e a *saída* a alterações no DOM (e, eventualmente, requisições enviadas servidores na internet). O app usado foi uma simples calculadora de IMC. O app foi feito e publicado na aula: http://150.165.85.16/~dalton/imc/

5. JavaScript

5.1: Introdução a JavaScript

1. javascript é mais Lisp e Smalltalk que Java
    a. sintaxe básica é de java
    b. semântica é muito mais lisp e smalltalk
    c. desconfie da intuição ao entender construções da linguagem

2. v0 de javascript foi projetada e implementada em 10 dias
    a. logo, inclui muitas más decisões de projeto
    b. que não podem ser eliminadas para ter compatibilidade

3. veremos principalmente aspectos de ES5 e ES6
    a. alguns aspectos mais antigos serão mencionados
    b. inovações de ES7, ES8 e ES9 também, quando relevante
    c. na prática, escolha a versão com base nos clientes