

.css: estilos

display: valores;

```
display: none; /* desabilita a exibição do elemento, sem afetar o layout. todos os elementos filhos também tem sua exibição desabilitada. o documento é renderizado como se o elemento não existisse */

display: inline; /* elementos em linha, como <span>. propriedades height e width não tem efeito nele */

display: block; /* elementos em bloco, como <p>. começa em uma nova linha, toma toda a largura */

display: list-item; /* gera uma caixa de bloqueio para o conteúdo e uma caixa embutida de item de lista separada */

display: inline-block; /* gera uma caixa de elemento de bloco que fluirá como se fosse uma única caixa embutida. o elemento em si se comportará como inline, porém pode receber valores de height e width */

display: flex; /* elemento se comporta como um elemento de bloco e apresenta seu conteúdo de acordo com o modelo flexbox */

display: inline-flex; /* elemento se comporta como um elemento embutido e apresenta seu conteúdo de acordo com o modelo flexbox */

display: grid; /* elemento se comporta como um elemento de bloco e apresenta seu conteúdo de acordo com o modelo de grade */

display: inline-grid; /* elemento se comporta como um elemento embutido e apresenta seu conteúdo de acordo com o modelo de grade */

display: contents; /* faz a caixa desaparecer, tornam os elementos filhos do próximo elemento acima do DOM */
```

flexbox

providencia um jeito mais eficiente de arranjar, alinhar e distribuir espaços entre itens em um container, mesmo quando seu tamanho é desconhecido ou dinâmico. é mais apropriado para os componentes de uma aplicação, e layouts de menor escala

a ideia principal por trás do layout flex é dar ao container a habilidade de alterar as alturas e larguras de seus itens (e order) para melhor preencher o espaço disponível – principalmente para acomodar a responsividade

um container flex expande seus itens para preencher o espaço livre disponível or diminui-los para prevenir excessos

mais importante, o layout flexbox possui uma direção-agnóstica, oposto aos layouts regulares – block, direção-vertical; inline, direção-horizontal. enquanto eles funcionam bem para páginas, lhes falta flexibilidade para o suporte de aplicações grandes e complexas (especialmente sobre mudança de orientação, tamanho, alongamento, etc)

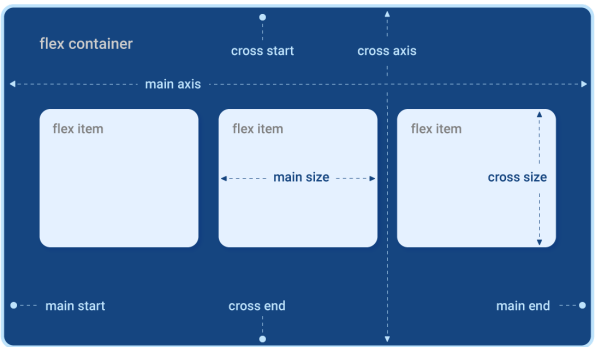
básico e terminologia

como flexbox é um módulo completo e não só uma propriedade única, envolve muitas coisas, inclusive seu próprio set de propriedades. algumas devem ser declaradas no container (elemento pai, conhecido como “flex container”) enquanto outras são declaradas nos elementos filhos (chamados de flex items)

flex container é o elemento que envolve sua estrutura. flex item são elementos filhos do flex container. eixos ou axes são as duas direções básicas que existem em um flex container: main axis, eixo principal, e cross axis, eixo transversal

```
/*
<div class="flex-container">
  <div class="flex-item">1</div>
  <div class="flex-item">2</div>
  <div class="flex-item">3</div>
</div>
*/

.flex-container {
  display: flex;
}
```



se um layout “padrão” é baseado em direções flow block e inline, o layout flex é baseado em direções flex-flow

itens são dispostos no main-axis (do main-start ao main-end) ou no cross-axis (do cross-start ao cross-end) – eixo principal ou transversal

- ▼ main axis (eixo principal)
 - ▼ de um flex container é o eixo primário em que ao longo dele serão inseridos os flex itens. não é necessariamente horizontal; depende da propriedade **flex-direction**
- ▼ main-start | main-end
 - os flex itens são inseridos dentro do container indo do start ao end
- ▼ main size (tamanho principal)
 - a altura e largura de um flex item, dependendo da direção do container, será o tamanho principal do item. a propriedade de tamanho principal de um flex item pode ser tanto width quanto height, dependendo de qual delas estiver na direção principal
- ▼ cross axis (eixo transversal)
 - o eixo perpendicular ao eixo principal, sua direção depende da direção do eixo principal
- ▼ cross-start | cross-end
 - linhas flex são preenchidas com itens e adicionadas ao container, começando pelo start do flex container ao lado end
- ▼ cross size
 - a altura e largura de um flex item, dependendo do que estiver na dimensão transversal, é o cross size do item. a propriedade cross size pode ser tanto width como height do item, o que estiver na transversal

flex vs inline-flex

flex faz com que o container se expanda ocupando toda a largura do layout, assim, os outros containers com o mesmo valor de **display** ficam um embaixo do outro, na direção vertical

`inline-flex` utiliza as mesmas características de exibição do `inline`, elementos em nível de linha, na horizontal, sem ocupar toda a largura do layout

propriedades flexbox

é importante saber quais as propriedades que são declaradas no flex container – por exemplo, uma div que irá conter os elementos a serem alinhados – e quais serão declaradas nos flex itens

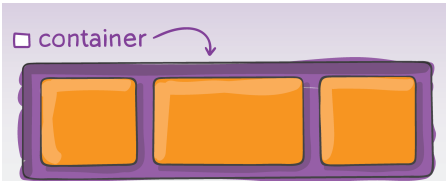
<https://codepen.io/luanalvesdev/pen/YzaaodX>

flex-container	flex-item
display: flex	order
flex-direction	flex-grow
justify-content	flex-shrink
flex-wrap	flex-basis
flex-flow	flex
align-items	align-self
align-content	
gap	

flex container (elemento-pai)

`display` → é o que define o flex container, inline ou block dependendo do valor declarado. coloca todos os elementos filhos diretos num contexto flex

```
.container {
  display: flex; /* ou inline-flex */
}
```



a propriedade `columns` não tem qualquer efeito sobre o flex container

`flex-direction` → estabelece o main axis, definido a direção que os flex itens serão alinhados no flex container. o flexbox é – com exceção do wrapping opcional – um conceito de layout de direção única. pensando nos flex itens como inicialmente posicionados ou em linhas horizontais ou em colunas verticais

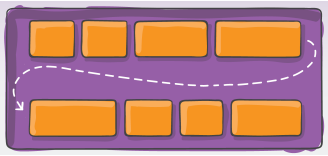
```
.container {
  flex-direction: row | row-reverse | column | column-reverse;
}
```



- ▼ row (padrão)
esquerda para direita em ltr – left to right. direita para esquerda em rtl – right to left
- ▼ row-reverse
direita para esquerda em ltr. esquerda para direita em rtl
- ▼ column
mesmo que `row`, mas de cima para baixo
- ▼ column-reverse
mesmo que `row-reverse`, mas de baixo para cima

`flex-wrap` → por padrão, flex itens vão tentar se encaixar em uma só linha. isso pode ser mudado, permitindo que os itens quebrem para uma linha seguinte conforme for necessário

```
.container {
  flex-wrap: nowrap | wrap | wrap-reverse;
}
```



- ▼ nowrap (padrão)
todos os itens flex estarão em uma única linha
- ▼ wrap
flex itens serão disposto em múltiplas linhas, de cima para baixo
- ▼ wrap-reverse

<https://codepen.io/team/css-tricks/pen/bEajLE/1ea1ef35d942d0041b0467b4d39888d3>

flex itens serão disposto em múltiplas linhas, de baixo para cima

flex-flow → propriedade *shorthand* – uma mesma declaração inclui vários valores relacionados a mais de uma propriedade, ex, **background** – que inclui **flex-direction** e **flex-wrap**. determina quais serão os eixos principal (main) e transversal (cross) do container

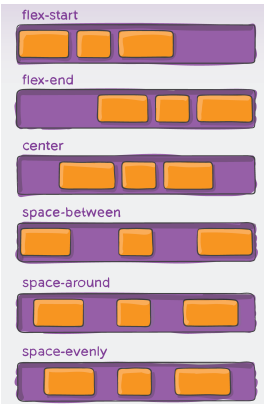
```
.container {
  flex-flow: column wrap;
}
```

justify-content → define o alinhamento ao longo do main axis. ajuda a distribuir o espaço livre que sobrar no container tanto quando os flex itens em uma linha são inflexíveis, quando são flexíveis mas já atingiram seu tamanho máximo. também exerce controlo sobre o alinhamento de itens quando eles ultrapassam o limite da linha

```
.container {
  justify-content: flex-start | flex-end | center | space-between | space-around | space-evenly | start | end | left | right ... + safe | unsafe;
}
```

o suporte dado pelos navegadores para estes valores é difuso. Por exemplo, space-between não tem suporte em nenhuma versão do Edge (até a elaboração deste tutorial). para tabelas detalhadas, consulte o MDN. os valores mais seguros são flex-start, flex-end e center

também existem duas palavras-chave adicionais que você pode usar em conjunto com estes valores: **safe** e **unsafe** (firefox). safe garante que, independente da forma que você faça esse tipo de posicionamento, não seja possível "empurrar" um elemento e fazer com que ele seja renderizado para fora da tela (por exemplo, acima do topo), de uma forma que faça com que o conteúdo seja impossível de movimentar com a rolagem da tela (o CSS chama isso de "perda de dados")

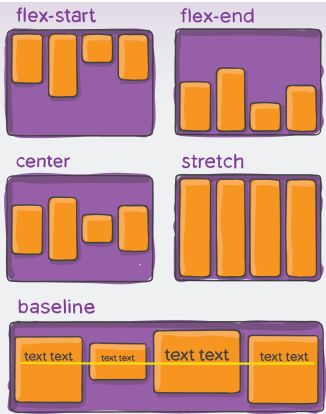


- ▼ flex-start (padrão)
os itens são alinhados junto à borda de início (start) de acordo com qual for a **flex-direction** do container
- ▼ flex-end
os itens são alinhados junto à borda final (end) de acordo com qual for a **flex-direction** do container
- ▼ start
os itens são alinhados junto à borda de início da direção do **writing-mode** (modo de escrita)
- ▼ end
os itens são alinhados junto à borda final da direção do **writing-mode** (modo de escrita)
- ▼ left
os itens são alinhados junto à borda esquerda do container, a não ser que isso não faça sentido com o **flex-direction** que estiver sendo utilizado. neste caso, se compartilha como **start**
- ▼ right
os itens são alinhados junto à borda direita do container, a não ser que isso não faça sentido com o **flex-direction** que estiver sendo utilizado. neste caso, se compartilha como **start**
- ▼ center
os itens são centralizados na linha
- ▼ space-between
os itens são distribuídos de forma igual ao longo da linha, o primeiro item junto à borda inicial da linha, o último à borda final
- ▼ space-around
os itens são distribuídos na linha com o mesmo espaçamento entre eles. visualmente, o espaço pode não ser igual, uma vez que todos os itens tem a mesma quantidade de espaço dos dois lados: o primeiro item vai ter somente uma unidade de espaço junto à borda do container, mas duas unidades de espaço entre ele e o próximo item, pois o próximo item também tem seu próprio espaçamento que está sendo aplicado
- ▼ space-evenly
os itens são distribuídos de forma que o espaçamento entre quaisquer dois itens da linha (incluindo os itens e as bordas) seja igual

align-items → define o comportamento padrão de como os flex itens são alinhados de acordo com o cross axis. de certa forma, funciona de forma similar ao **justify-content**, porém no eixo transversal (perpendicular ao principal)

```
.container {
  align-items: stretch | flex-start | flex-end | center | baseline | first baseline | last baseline | start | end | self-start | self-end + ... safe | unsafe;
}
```

os modificadores **safe** e **unsafe** podem ser usados em conjunto com todas essas palavras-chave (confira o suporte de cada navegador) e servem para prevenir qualquer alinhamento de elementos que faça com que o conteúdo fique inacessível (por exemplo, para fora da tela)



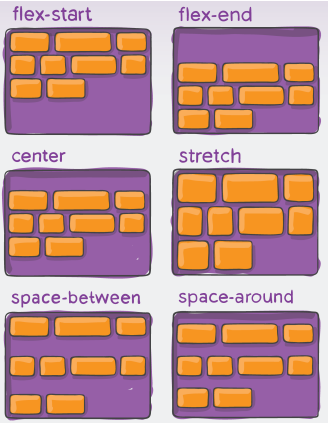
- ▼ stretch (padrão)
estica os itens para preencher o container, respeitando o **min-width/max-width**
- ▼ flex-start / start / self-start

- itens são posicionados no inicio do eixo transversal. a diferença entre eles é sutil e diz respeito às regras de `flex-direction` ou `writing-mode`
- ▼ flex-end / end / self-end
 - itens são posicionados no final do eixo transversal. novamente, a diferença é sutil e diz respeito às regras de `flex-direction` vs `writing-mode`
- ▼ center
 - itens são centralizados no eixo transversal
- ▼ baseline
 - itens são alinhados de acordo com suas baselines

align-content → **organiza** as linhas de um flex container quando há espaço extra no eixo transversal, similar ao modo como `justify-content` alinha itens individuais dentro do eixo principal. essa propriedade não tem efeito quando há somente uma linha de flex itens no container (i.e. `flex-wrap: no-wrap;`)

```
.container {
  align-content: flex-start | flex-end | center | space-between | space-around | space-evenly | stretch | start | end | baseline | first baseline | last baseline
}
```

esta propriedade não tem efeito quando há somente uma linha de flex items no container
os modificadores `safe` e `unsafe` podem ser usados em conjunto com todas essas palavras-chave (confira o suporte de cada navegador) e servem para prevenir qualquer alinhamento de elementos que faça com que o conteúdo fique inacessível (por exemplo, para fora da tela)



- ▼ normal (padrão)
 - itens são alinhados em suas posições primarias como se nenhum valor estivesse declarado
- ▼ flex-start / start
 - itens são alinhados com o inicio do container. o valor `flex-start` (com maior suporte dos navegadores) se guia pela `flex-direction`, enquanto `start` se guia pela direção do `writing-mode`
- ▼ flex-end / end
 - itens são alinhados com o final do container. o valor `flex-end` (com maior suporte dos navegadores) se guia pela `flex-direction`, enquanto `end` se guia pela direção do `writing-mode`
- ▼ center
 - itens são centralizados no container
- ▼ space-between
 - itens são distribuídos igualmente; a primeira linha junto ao inicio do container e a última junto ao final
- ▼ space-around
 - itens distribuídos igualmente com o mesmo espaçamento entre cada linha
- ▼ space-evenly
 - itens distribuídos igualmente com o mesmo espaço entre eles
- ▼ stretch
 - itens em cada linha esticada para ocupar o espaço remanescente entre elas

gap, row-gap, column-gap → controla explicitamente o espaço entre os flex itens. aplica espaçamento *somente entre os itens*, não nas bordas externas

```
.container {
  display: flex;
  ...
  gap: 10px;
  gap: 10px 20px; /* row-gap column gap */
  row-gap: 10px;
  column-gap: 20px;
}
```

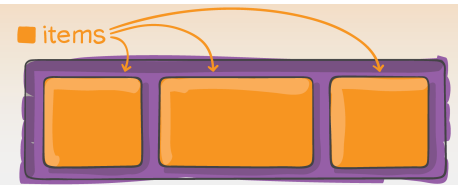
o seu comportamento pode ser visto como um vão mínimo, se o vão é de alguma forma maior (por causa de algo como `justify-content: space-between;`) então o gap só iria ter efeito se aquele espaço acabasse sendo menor. não é exclusivo do flexbox, `gap` também funciona no grid e em layouts de multi-colunas



flex itens (elementos-filhos)

onde existem um elemento-pai com propriedade `flex` (o flex container), é possível atribuir propriedades flex específicas também para os elementos-filhos (flex item)

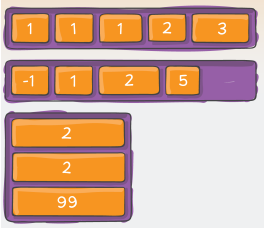
```
<div class="container">
  <div>1</div> /* flex item */
</div>
```



order → determina a ordem em que os elementos aparecerão. por padrão, os flex itens são dispostos na ordem da sua fonte (código)

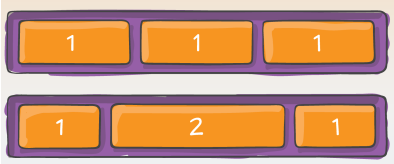
```
.item {
  order: 5; /* default is 0 */
}
```

itens com a mesma ordem revertem para ordem da fonte. aceita valores negativos, já que a contagem começa pela posição do item selecionado [0], sendo as posições a sua direita positivas [1] e a sua esquerda negativas [-1] leitores de tela irão ignorar a propriedade e lerão de acordo com a order no código html



flex-grow → define a habilidade de um flex item de crescer se necessário. aceita um valor numérico sem unidade que serve como proporção. dita a quantidade de espaço disponível no container que será ocupado pelo item

```
.item {
  flex-grow: 2; /* default 0 */
}
```



se todos os itens tiverem **flex-grow** definido em 1, o espaço remanescente no container será distribuído de forma igual entre todos. se um dos itens tem o valor de 2, vai ocupar o dobro de espaço no container com relação aos outros (ou pelo menos tentará) valores negativos não são aceitos pela propriedade

flex-shrink → define a habilidade de um flex item de encolher, caso necessário

```
.item {
  flex-shrink: 1; /* default 1 */
}
```

números negativos são inválidos



flex-basis → define o tamanho padrão de um elemento antes que o espaço remanescente do container seja distribuído. pode ser um comprimento (20%, 5rem, etc) ou uma palavra chave

```
.item {
  flex-basis: 50px; /* default auto */
}
```



a keyword **auto** significa “olhe para minhas propriedades **height** ou **width**” (o que era feito pela keeyword **main-size**, antes de ser depreciada) **content** significa “estabeleça o tamanho com base no conteúdo interno do item” – ainda não tem suporte amplo, então não é fácil de ser testada, assim como suas relacionadas: **max-content**, **min-content** e **fit-content** com o valor de 0, o espaço extra ao redor do conteúdo não é considerado. com o valor de **auto**, o espaço extra é distribuído com base no valor de **flex-grow** do item

flex → é a propriedade *shorthand* para **flex-grow**, **flew-shrink** e **flex-basis**, combinadas. o segundo e terceiro parâmetros são opcionais. o é **0 1 auto**, mas se você definir com apenas um número, é equivalente a **0 1**

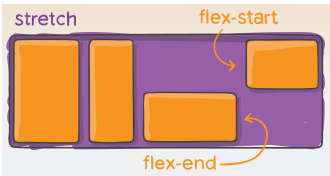
```
.item {
  flex: none | [ <'flex-grow'> <'flex-shrink'>? || <'flex-basis'> ]
}
```

é **recomendo usar esta propriedade shorthand** ao invés de definir cada uma das propriedades separadamente

align-self → permite que o alinhamento padrão (ou o especificado pelo **align-items**) seja sobscrito para flex itens individuais

```
.item {
  align-self: auto | flex-start | flex-end | center | baseline | stretch;
}
```

rever **align-items** para entender quais são os possíveis valores



!!!

css só enxerga hierarquia pai-filho; não vai aplicar as propriedades flex para elementos que não estejam diretamente relacionados; para que as propriedades funcionem nos elementos-filhos, os pais devem ter propriedade **display: flex;** as propriedades **float**, **clear** e **vertical-align** não tem efeito em flex-itens

prefixos flexbox

para que tenha suporte para o maior número de navegadores possíveis, flexbox faz uso de alguns prefixos. não há só inclui propriedades prefixadas, mas também propriedades inteiramente diferentes e nomes de valores. isso é porque o flexbox spec mudou com o tempo, criando “olds”, “tweener” e “new” versões

um jeito de lidar com isso é escrevendo na nova (e na final) sintaxe e rodar pelo css por um autoprefixer, o que lida bem com fallbacks. alternativamente, aqui está um sass@mixin para ajudar em alguns prefixos, o que dá uma ideia do que se precisa fazer

```
@mixin flexbox() {
  display: -webkit-box;
  display: -moz-box;
  display: -ms-flexbox;
  display: -webkit-flex;
  display: flex;
}

@mixin flex($values) {
  -webkit-box-flex: $values;
  -moz-box-flex: $values;
  -webkit-flex: $values;
  -ms-flex: $values;
  flex: $values;
}

@mixin order($val) {
  -webkit-box-ordinal-group: $val;
  -moz-box-ordinal-group: $val;
  -ms-flex-order: $val;
  -webkit-order: $val;
  order: $val;
}

.wrapper {
  @include flexbox();
}

.item {
  @include flex(1 200px);
  @include order(2);
}
```

exemplo de uma centralização perfeita

```
.parent {
  display: flex;
  height: 300px; /* Or whatever */
}

.child {
  width: 100px; /* Or whatever */
  height: 100px; /* Or whatever */
  margin: auto; /* Magic! */
}
```

grid


é mais apropriado para layouts de maior escala

permite declarar a posição, tamanho e espaçamento de um elemento, criar “lacunas” ou “buracos” que serão preenchidos pelas tags html

básico

divididos em grid container – que aloca os elementos do layout – é o elemento-pai e grid item – os próprios elementos – são os filhos diretos. a maioria das suas propriedades é declarada no container

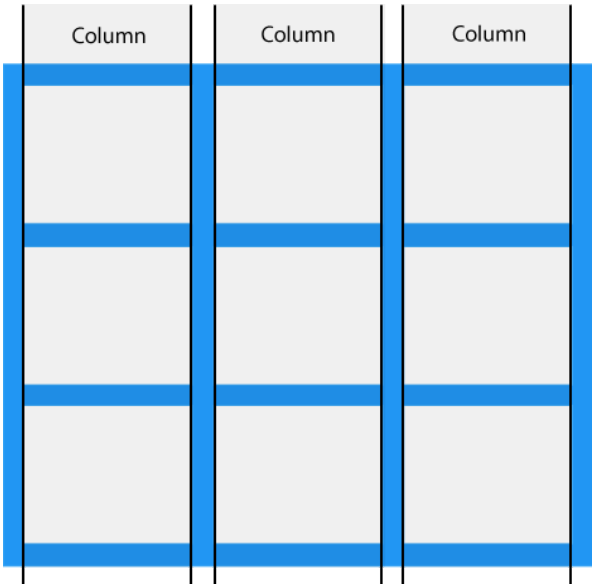
```
.grid-container {
  display: grid; /* ou inline-grid */
}
```



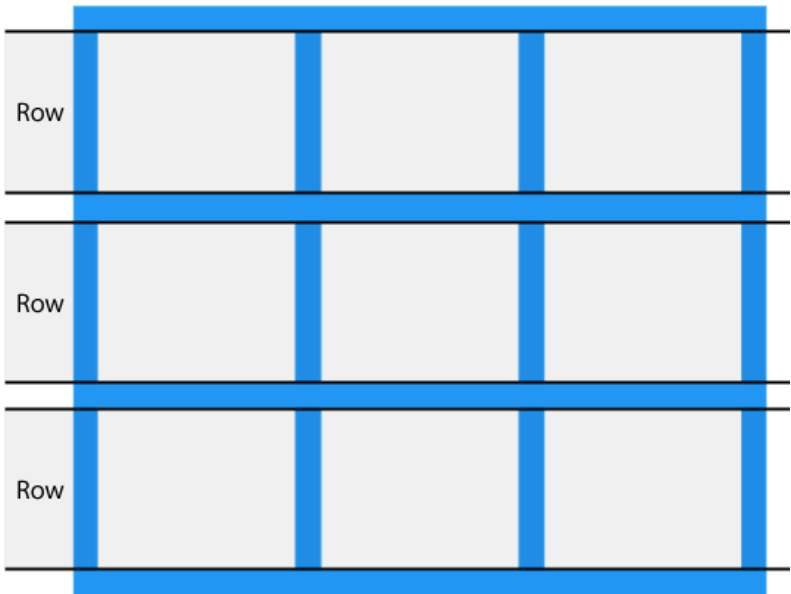
```
1 * <div id="grid-container">
2 *   <div class="grid-item">Item 1</div>
3 *   <div class="grid-item">Item 2</div>
4 *   <div class="grid-item">Item 3</div>
5 * </div>
```

todos os filhos diretos do grid container se transformarão automaticamente em grid itens

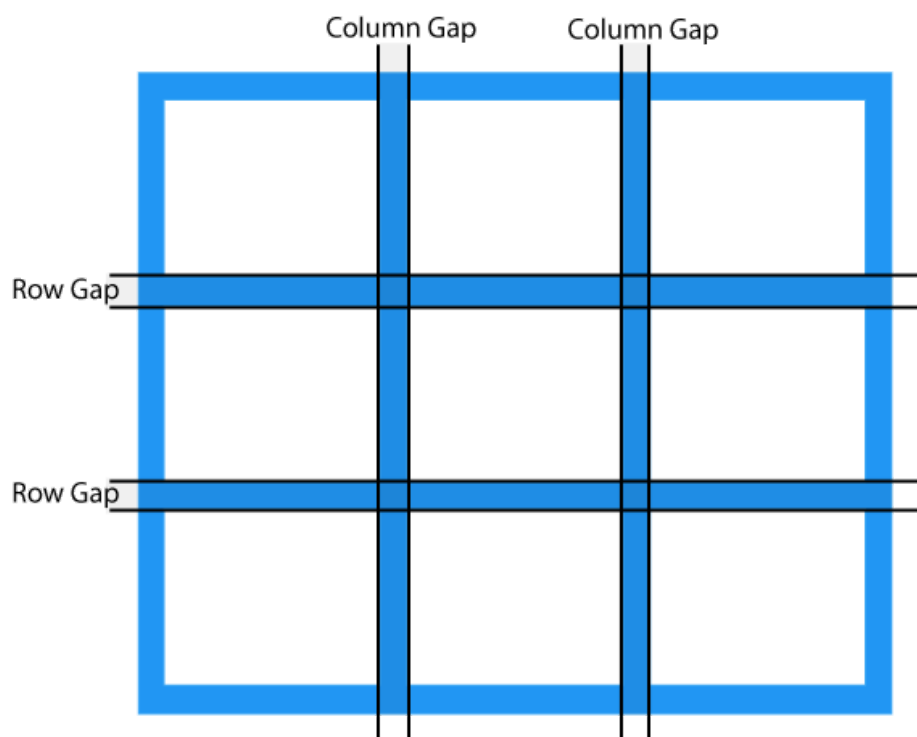
grid columns → as linhas verticais de grid itens



grid rows → as linhas horizontais de grid itens



grid gaps → os espaços entre as colunas e linhas



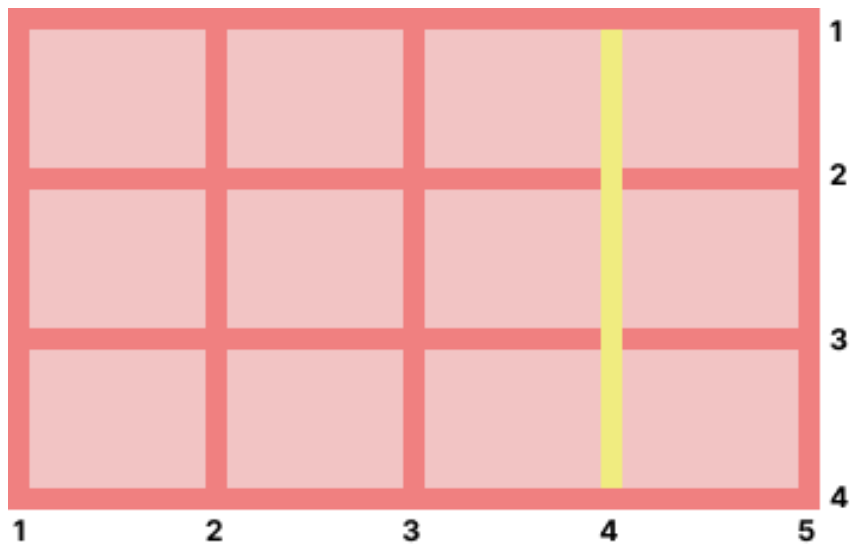
usando as propriedades determinadas, pode-se ajustar cada um

```
.grid-container {
  display: grid;
  column-gap: 50px;
}
```

```
.grid-container {
  display: grid;
  row-gap: 50px;
}
```

```
.grid-container {
  display: grid;
  gap: 50px 100px; /* propriedade shorthand que declara ambos row / columns */
}
```

grid lines → as linhas entre as colunas são chamadas de **column lines**, as linhas entre as linhas são chamadas de **row lines**



referenciando números de linhas quando posicionar grid itens no grid container

grid item na column line 1, termina na column line 3

```
grid-column-start: 1;
grid-column-end: 3;
grid-column: 1/3;
```

grid item na row line 1, termina na row line 3

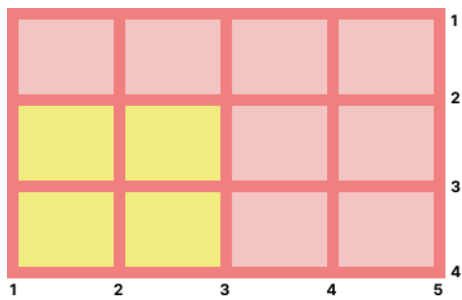
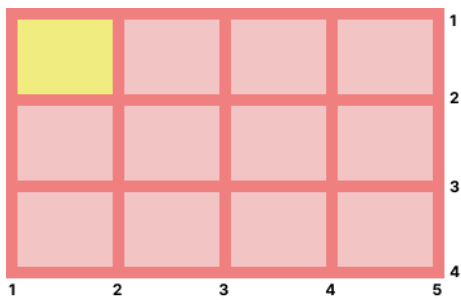
```
grid-row-start: 1;
grid-row-end: 3;
grid-row: 1/3
```

propriedade *shorthand* → `grid-row-start / grid-column-start / grid-row-end / grid-column-end`

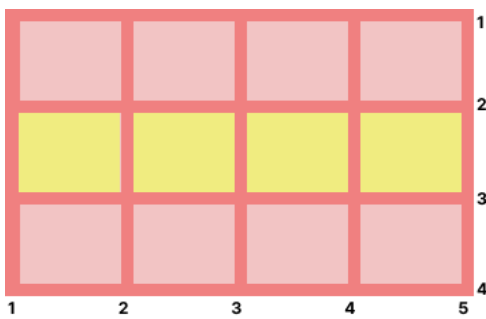
```
grid-area: 1 / 1 / 3 / 3;
```

grid cell → é um espaço entre 4 grid lines que se cruzam. menor unidade no grid, muito parecido com uma célula de tabela

grid area → conjunto de uma ou mais grid cells que formam uma área que pode ser nomeada



grid track → trilha de grade é o espaço entre duas grid lines adjacentes. pode ser uma linha completa ou uma coluna completa



imagine um layout com um cabeçalho, um menu lateral, o conteúdo principal e um rodapé. montando o html ficaria mais ou menos assim:

```
<header>Cabeçalho</header>
<aside>Menu lateral</aside>
```

```
<main>Conteúdo principal</main>
<footer>Rodapé</footer>
```

montar a estrutura das tags é a parte fácil, porém, agora precisa posiciona-los da seguinte maneira:

- Header: Em cima da página
- Aside: Na parte esquerda da página
- Main: Na parte direita da página
- Footer: Em baixo da página



para chegar nesse resultado, pode-se usar o **float** ou **flexbox**, medindo o tamanho de cada elemento, adicionado um pouco de **margin** e **padding**... ou pode-se usar o layout grid

para desenvolver um layout grid, primero devesse o declarar dentro do container que irá envolver nossos elementos

```
<header class="o-header">Header</header>
<aside class="o-aside">Aside</aside>
<main class="o-main">Main</main>
<footer class="o-footer">Footer</footer>
```

```
body {
  display: grid;
}
```

o posicionamento desses elementos é feito de uma forma “desenhada” no css, que será feito desta maneira

```
body {
  display: grid;
  grid-template-areas:
    "header header header"
    "aside main main"
    "footer footer footer";
}
```

entendendo o template areas

com a propriedade **css-template-areas** “desenhamos” o layout, assim informando como e por quais elemenots ele vai ser composto:

- “header header header” → declara que a primeira linha vai ser composta por um **header**
- “aside main main” → declara que a segunda linha vai ser composta por um **aside** na esquerda e um **main** na direita
- “footer footer footer” → declara que a terceira e última linha vai ser composta por um **footer**

o **header** e o **footer** foram declarados 3x, pois estamos trabalhando com um layout de 3 colunas, isso deve-se a segunda linha do código. como o resultado final deseja que o **main** seja 2x maior que o **aside**, o declaramos 2 vezes, criando então um coluna extra que será ocupada pela extensão desses elementos

informado onde os elementos devem ficar

para dizer aos elementos ondem se posicionar, ou seja, qual “lacuna” é de cada um, precisa-se ir em cada um deles para declara-los

```
.o-header {
  grid-area: header;
}

.o-aside {
  grid-area: aside;
}

.o-main {
  grid-area: main;
}

.o-footer {
  grid-area: footer;
}
```

```
.o-header, .o-aside, .o-main, .o-footer {
  background: #BC20E2;
  color: #FDFDFD;
}
```

espaçamento e tamanhos dos elementos

para se definir espaçamento entre elementos, usa-se a propriedade **grid-gap**, que aceita as medidas usuais, como rem e px

```
body {
  grid-gap: 1rem;
}
```

para definir o tamanho de cada row (linha) ou column (coluna), precisa-se declara-las no template

```
body {
  grid-template-columns: auto auto auto;
  grid-template-rows: auto 100vh auto;
}
```

<https://codepen.io/olwr/pen/poVZbMQ>

novamente, são informados 3 valores, pois temos um layout de 3 colunas e 3 linhas, veja também que apenas para a segunda linha declaramos o valor `100vh` que corresponde a altura total da **viewport**, as demais linha são automáticas

por padrão, o valor será `auto`

repeat

função `repeat()` recebe como parâmetro dois argumentos, o primeiro é o **repeat count**, que se refere a quantas vezes o valor irá ser repetir e o segundo é o **track** que é o valor em si que será repetido. O repeat count pode ser um valor inteiro de 1 ou mais, como por exemplo no código: `repeat(3, auto)`, 3 é o valor do repeat count. Mas também o valor pode ser palavras-chaves, como `auto-fill` ou `auto-fit`

auto-fill

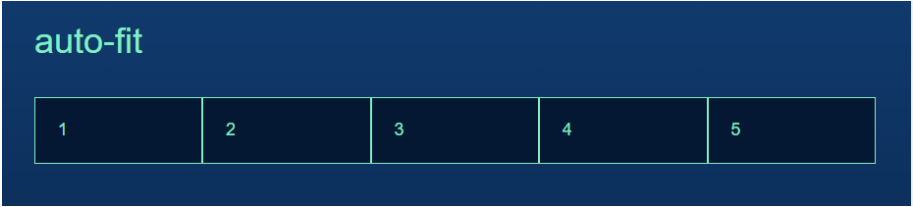
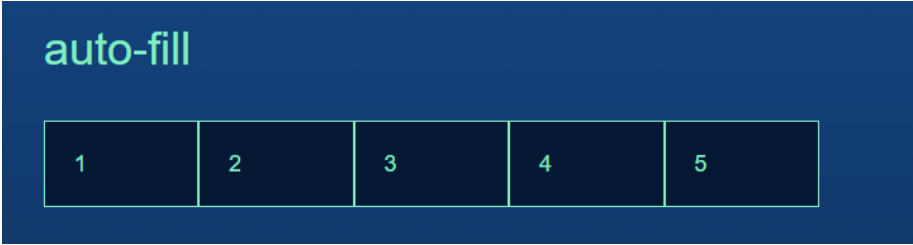
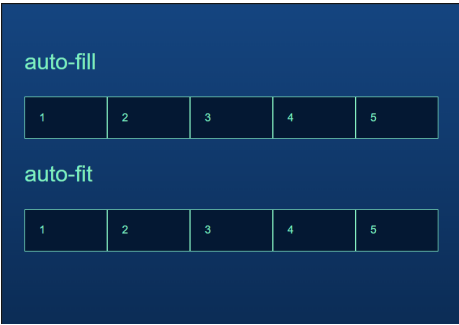
preenche a linha do Grid Container com novas colunas sempre que há espaço para caber mais uma, o objetivo é preencher a linha com o máximo de colunas possível. Caso o viewport seja maior do que a largura do Grid container e número de colunas existentes, esse será o resultado:

dessa forma, haverá um espaço sobrando do lado direito, as colunas não expandirão o tamanho para preencher esse vazio

auto-fit

ajusta o tamanho das colunas para ocupar o tamanho total do espaço disponível do container, expandindo-as para que não fique nenhum espaço sobrando. Independente do tamanho da tela, esse será o resultado:

o grid container possui uma largura de aproximadamente 600px, note que nesta mesma largura de tela, não há diferença visual entre os valores `auto-fill` e `auto-fit`



mas a partir do momento que aumenta o tamanho da viewport, começa a aparecer a diferença dos repeat count

