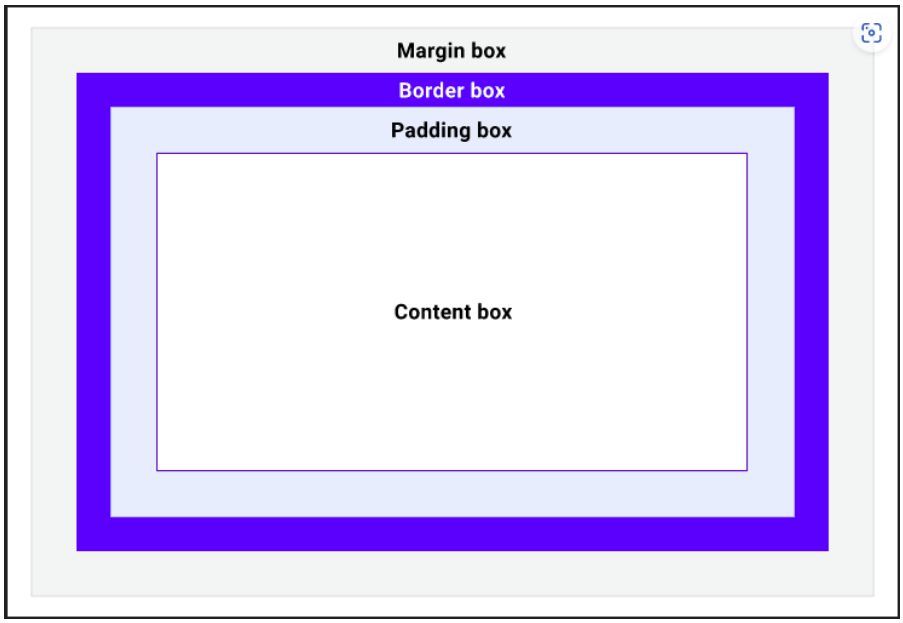
NOTES

▼ CSS

▼ 1. Box Model

- A really important thing to remember when writing CSS, or working on the web as a whole, is that everything displayed by CSS is a box. Whether that's a box that uses `border-radius` to look like a circle, or even just some text: the key thing to remember is that it's all boxes.

Area of box model:



- **Margin: It** is the space around your box, defined by the `margin` rule on your box. Properties such as `outline` and `box-shadow` occupy this space too because they are painted on top, so they don't affect the size of our box.
- **Border:** The **border box** surrounds the padding box and its space is occupied by the `border` value.
- **Padding:** The **padding box** surrounds the content box and is the space created by the `padding` property. Because padding is inside the box, the background of the box will be visible in the space that it creates. If our box has overflow

rules set, such as `overflow: auto`
or `overflow: scroll`, the scrollbars will occupy this space too.

- **Content:** It is the area where content lies.

- if we are in a normal flow, a `<div>` element's default `display` value is `block`, a `<li>` has a default `display` value of `list-item`, and a `<span>` has a default `display` of `inline`. display block cha bhane width ra height applicable hudaina.
- An Inline element has block margin, but other won't respect it. So we use `inline-block`. A `block` item will, by default, fill the available **inline space**, whereas a `inline` and `inline-block` elements will only be as large as their content.
- By default, all elements have the following user agent style: `box-sizing: content-box;`. It means when when we we set dimensions such as width and height , it is applied to **content box.** Also padding and border is applied to content box's size.

```
.box {
  width: 350px;
  height: 150px;
  margin: 10px;
  padding: 25px;
  border: 5px solid black;
}
```

The *actual* space taken up by the box will be 410px wide (350 + 25 + 25 + 5 + 5) and 210px high (150 + 25 + 25 + 5 + 5).

Alternate CSS Box model: `box-sizing: border-box`

```
// for same css above
```

The width of the box will be only 350px.

▼ 2. Selectors

- To apply the CSS we need to select the element. CSS provides with number of different ways to explore them.

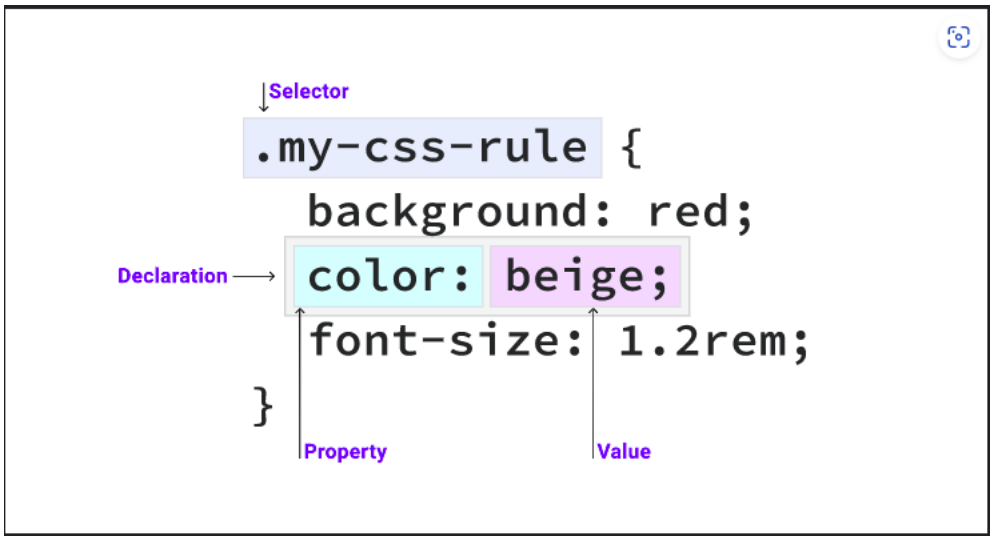
Example: if we want to select the first paragraph element in a article section we can:

```
<article>
<p>text1</p>
<p>text2</p>
</article>
```

```
article p: first-of-type{
color: red;
}
```

In this way we can select 1st paragraph and apply the properties.

Part of a CSS rule.



Simple selectors

- **universal Selectors:** It causes all element to have color red.

```
*{
color: red;
}
```

- **Type Selector:** It matches a HTML element directly.

```
nav{
margin: 4px;
}
```

- **Class Selector:** HTML element can have multiple elements with same class name. The class selector matches any element that has class applied to it.

```
// class is selected using "."
.class_example{
color: green;
}
```

A HTML element can have multiple classes.

```
<p class="class_A class_B" > </p>
```

- **ID Selector:** HTML element can have only one id unique id. No two element shall have same id name.

```
// id is selected using "#"
#id_eg{
color:red;
}
```

- **Attribute Selector:** We can look for elements that have a certain HTML attribute, or have a certain value for an HTML attribute.

```
// attribute is selected by wrapping the attribute in square bracket
[data-type] {       // this will apply to all the attribute data-type
  color: red;
}
```

```
<div data-type="primary"></div>
```

The attribute can be anything like “href” .

- **Grouping Selectors:** We can group multiple selectors by separating them with comma.

```
strong,
em,
.my-class,
[lang] {
  color: red;
}
```

Pseudo-classes and pseudo-elements

- `CSS provides useful selector types that focus on specific platform state, like when an element is hovered, structures *inside* an element, or parts of an element.

- **Pseudo Classes:** HTML elements find themselves in various states, either because they are interacted with, or one of their child elements is in a certain state. for example: hover.

```
/* Our link is hovered */
a:hover {
  outline: 1px dotted green;
}

/* Sets all even paragraphs to have a different background */
p:nth-child(even) {
  background: floralwhite;
}
```

- **Pseudo element:** It is different from pseudo-classes because instead of responding to platform state, they act as if they are inserting new element with CSS. We use `::` .

  But older version browser used to support pseudo-element with `:`

```
.my-element::before {
  content: 'Prefix - ';
}
```

As in the above demo where you prefixed a link's label with the type of file it was, you can use the `::before` pseudo-element to insert content at the start of an the `::after` pseudo-element to insert content at the end of an element.

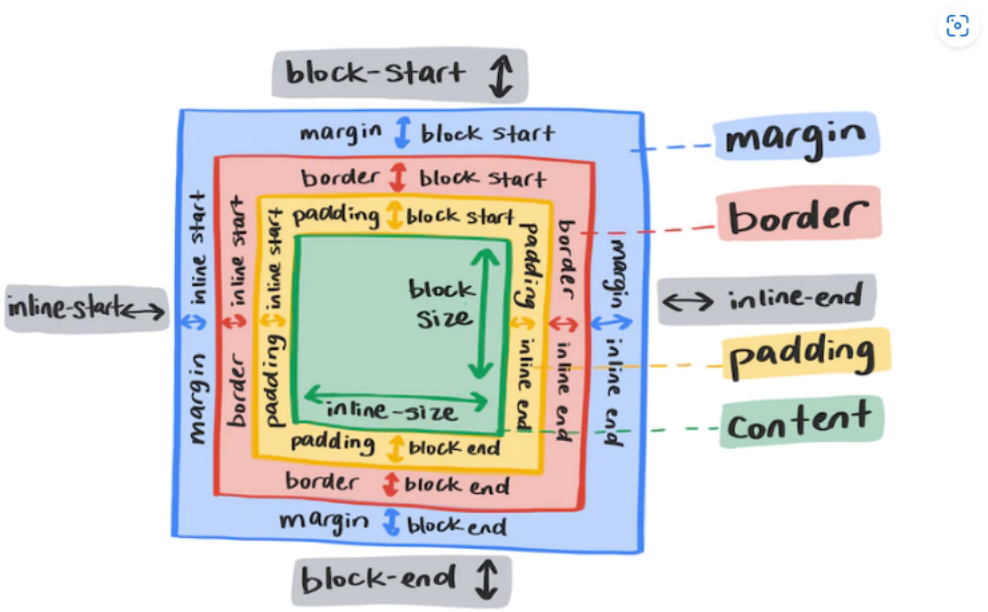
▼ 3. Layout

- The display property in CSS does two things, determines if the box it is applied to acts as inline or block.

```
.layout{
display: inline;
}
```

Inline elements behave like words in a sentence. They sit next to each other in the inline direction.

We can’t set explicit height and width on inline elements.




```
.layout{
display: block;
}
```

Block elements don't sit alongside each other. They create a new line for themselves. Unless changed by other CSS code, a block element will expand to the size of the inline dimension.

```
.layout{
display: flex;
```

```
}
```

The `display` property also determines how an element's children should behave. For example, setting the `display` property to `display: flex` makes the box a block-level box, and also converts its children to flex items.

▼ Flex-box

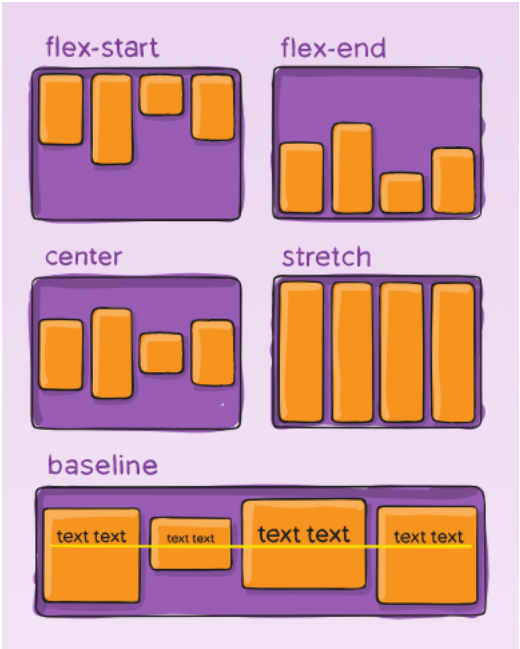
- Flex box is a layout manager for one-dimensional layouts. Layout across single axis. By default, flexbox will align the element's children next to each other, in the inline direction, and stretch them in the block direction, so they're all the same height.

```
.layout{
display: flex;
}
```

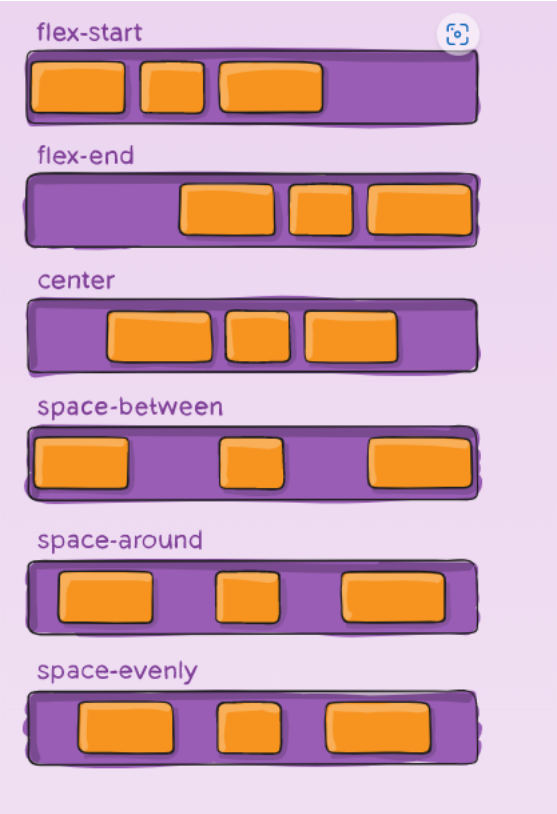
Items will stay on the same axis and not wrap when they run out of space. Instead they will try to squash onto the same line as each other. This behavior can be changed using the `align-items`, `justify-content` and `flex-wrap` properties.

▼ Properties

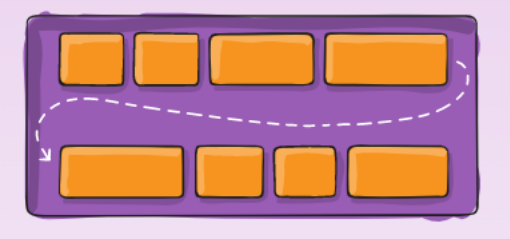
Align Items properties:



Justify Content Properties:



Flex Wrap Properties:



By default, flex items try to fit onto one line. We can change and allow items to wrap as needed with properties like:

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

- nowrap (default): all flex items will be in one line.
- wrap: flex items will wrap onto multiple lines, from top to bottom.
- wrap-reverse: flex items will wrap onto multiple lines from bottom to top.

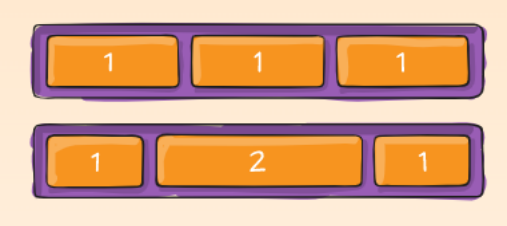
- flex-box also converts child elements to flex items, means we can write rules on how they behave inside a flex container. We can change alignment, order and justification on an individual item.

```
.element{
flex: 1 0 auto;
}
```

The `flex` property is shorthand for `flex-grow`, `flex-shrink`, `flex-basis`. Above example can be expanded line:

```
.element{
flex-grow: 1;
flex-shrink: 0;
flex-basis: auto;
}
```

- **flex-grow**



- **flex-shrink**

This defines the ability for a flex item to shrink if necessary.

- **flex-basis**

This defines the default size of an element before the remaining space is distributed.

1. `flex-grow`: This property determines how much the flex item will grow relative to other items in the flex container. When `flex-grow` is set to `1`, the item will take up as much available space as possible along the main axis.
2. `flex-shrink`: This property determines how much the flex item will shrink relative to other items in the flex container. When `flex-shrink` is set to `1`, the item will shrink if there is not enough space available along the main axis.
3. `flex-basis`: This property sets the initial size of the flex item before any remaining space is distributed. When `flex-basis` is set to `0` or `auto`, the item will take up its natural size.

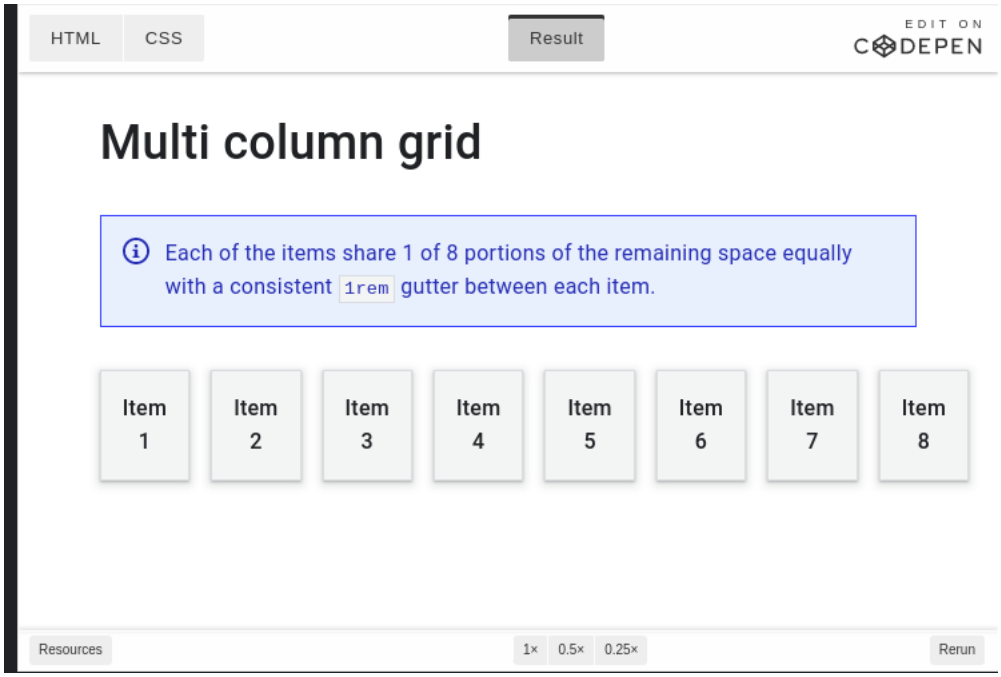
▼ Grid

- Grid is similar to flex-box but works for multi-axes layouts instead of single-axis layouts.

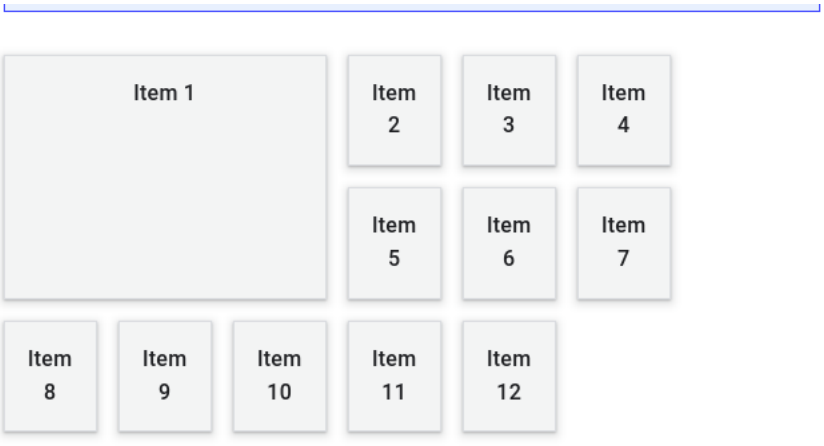
  Grid enables to write layout rules on an element that has `display:grid;` and introduces new primitives for layout styling. such as `repeat()` `minmax().`

One useful grid unit is `fr` unit that is a fraction of remaining space, we can build traditional 12 column grids.

```
.element{
display: grid;
grid-template-columns: repeat(12, 1fr);
gap: 1rem;
}
```



Above examples shows elements in single axis. Flexbox treats items as a group whereas grid gives precise control over their placement in two dimensions.




```
// we can define a item to take certain row and column space

.element : first-child{
grid-row: 1/3;
```

```
grid-column: 1/4;
}
```

▼ Sizing

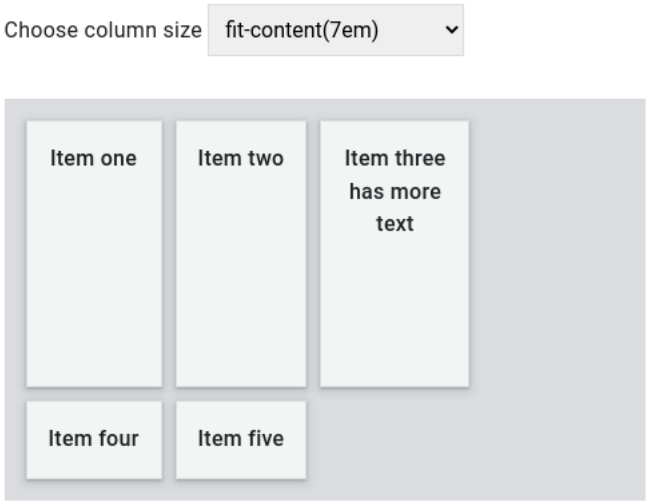
These keywords are defined in the Box Sizing specification and add additional methods of sizing boxes in CSS, not just grid tracks.

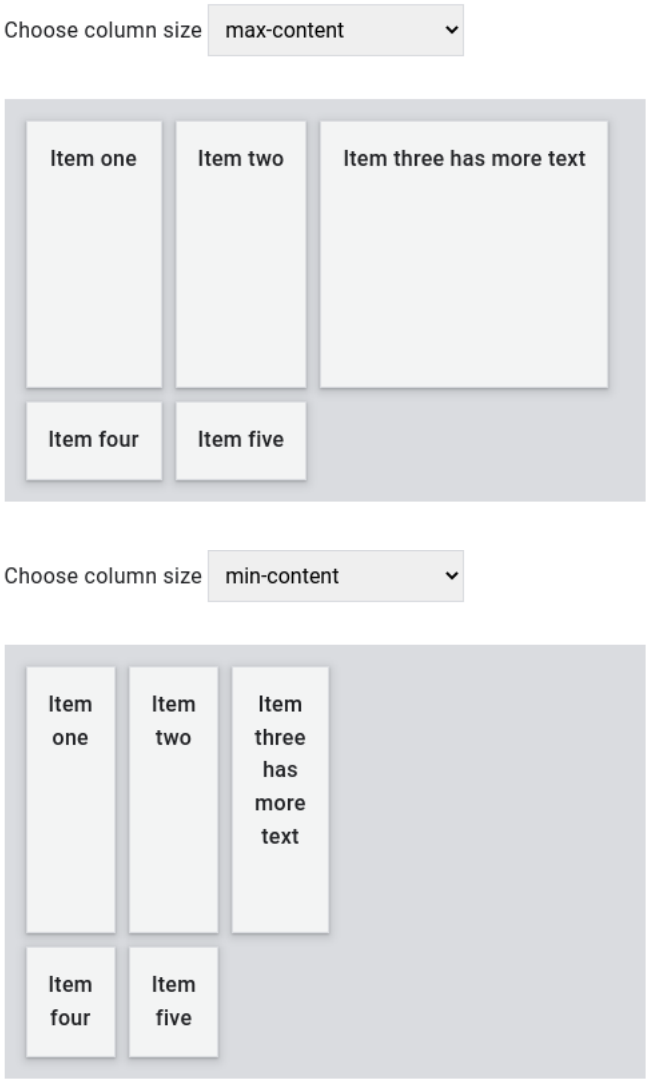
- `min-content`
- `max-content`
- `fit-content()`

The `min-content` keyword will make a track as small as it can be without the track content overflowing. Changing the example grid layout to have three column tracks all at `min-content` size will mean they become as narrow as the longest word in the track.

The `max-content` keyword has the opposite effect. The track will become as wide enough for all of the content to display in one long unbroken string. This might cause overflows as the string will not wrap.

The `fit-content()` function acts like `max-content` at first. However, once the track reaches the size that you pass into the function, the content starts to wrap. So `fit-content(10em)` will create a track that is less than 10em, if the `max-content` size is less than 10em, but never larger than 10em.



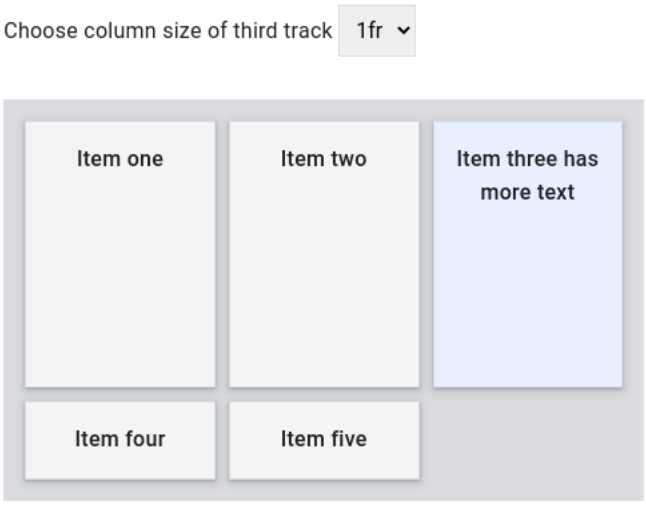


▼ fr unit

- There is also a special sizing method which only works in grid layout. This is the `fr` unit, a flexible length which describes a share of the available space in the grid container.
- The `fr` unit works in a similar way to using `flex: auto` in flex-box. It distributes space after the items have been laid out. Therefore to have three columns which all get the same share of available space:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
}
```

- Using different values for the fr unit will share the space in proportion. Larger values getting more of the spare space.



▼ The `minmax()` function

- We can set minimum and maximum size for track. above fr example can be written using minmax() as `minmax(auto, 1fr)` . Grid looks at the intrinsic size of the content, then distributes available space after giving the content enough room.
- To force a track to take an equal share of the space in the grid container minus gaps use minmax. Replace `1fr` as a track size with `minmax(0, 1fr)`. This makes the minimum size of the track 0 and not the min-content size. Grid will then take all of the available size in the container, deduct the size needed for any gaps, and share the rest out according to your fr units.

▼ repeat()

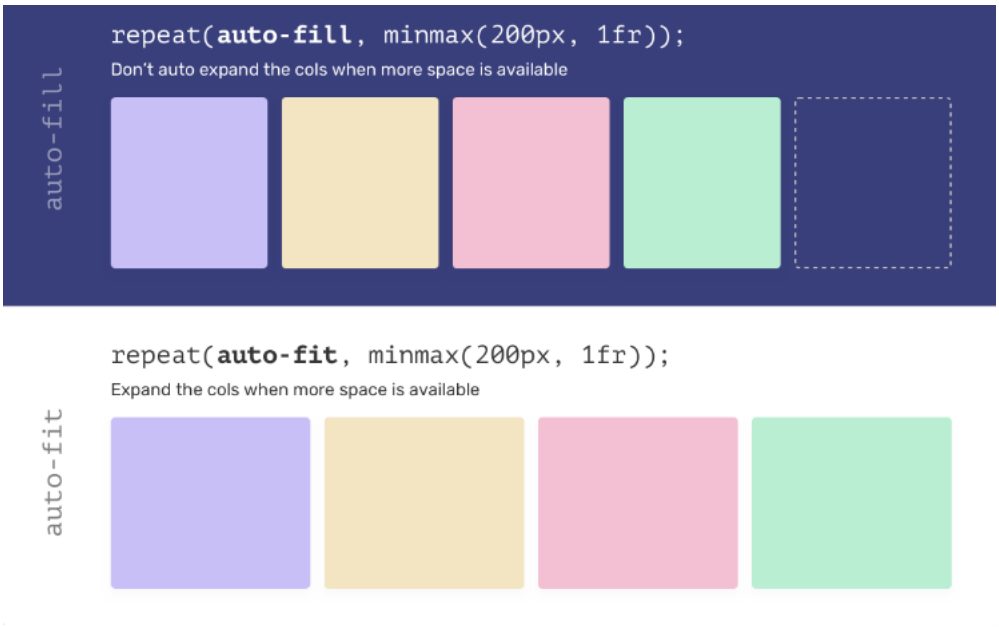
```
.container {
    display: grid;
    grid-template-columns: repeat(12, minmax(0,1fr));
}
```

The `repeat()` can be used to repeat any section of track listing. We can repeat pattern of tracks.

▼ auto-fill and auto-fit

- If we don't want to specify the number of column tracks, but instead want to create as many as will fit in our container, we can use this property.

  We can achieve this with `repeat()` and `auto-fill` or `auto-fit` keywords.



▼ Positioning with gird-template area

```
.container {
  display: grid;
  grid-template-areas:
    "header header"
    "sidebar content"
    "footer footer";
  grid-template-columns: 1fr 3fr;
  gap: 20px;
}

header {
  grid-area: header;
}

article {
  grid-area: content;
}

aside {
  grid-area: sidebar;
}

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

Rules:

1. To span over two cells repeat name
2. to leave cell empty we use `.`
3. area must be rectangular
4. area can't be repeated in different locations

▼ Grid-Simplified

▼ Display

- grid: generated block-level grid
- inline-grid: inline-level grid

▼ Grid-template-column | row

- defines the column and rows with space separated values.
- **values:**
  - `<track-size>` can be length, percentage, or a fraction of the free space in grid using the fr unit.
  - `<line-name>` is the arbitrary name of our choosing

```
.container {
  grid-template-columns: ...  ...;
```

```
  /* e.g.
      1fr 1fr
      minmax(10px, 1fr) 3fr
      repeat(5, 1fr)
      50px auto 100px 1fr
  */
  grid-template-rows: ... ...;
  /* e.g.
      min-content 1fr min-content
      100px 1fr max-content
  */
}
```

- Grid lines have positive number of assignments from start where (-1) is for the last item.

- The `fr` allows us to set the size of track as fraction of free space in the grid container.

```
.container{
grid-template-columns: 1fr 1fr 1fr;
}
```

here, each item is set to the 1/3 of the total width of the grid container.

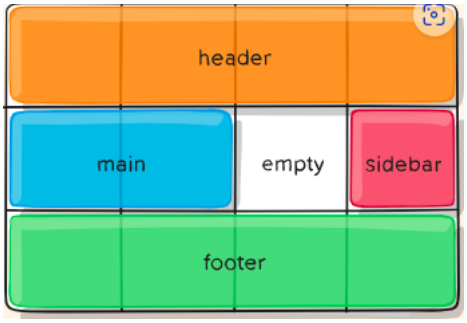
```
.container{
grid-template-columns: 1fr 1fr 50px 1fr;
}
```

but, here the amount of free space available to fr doesn't include 50px, the total space available for fr is total width minus 50px.

▼ Grid template areas

- referencing the name of area can make it easy to place the items in the grid layout.

```
.container {
  display: grid;
  grid-template-columns: 50px 50px 50px 50px;
  grid-template-rows: auto;
  grid-template-areas:
    "header header header header"
    "main main . sidebar"
    "footer footer footer footer";
}

.item1{
grid-area: header;
}
.item2{
grid-area: main;
}
.item3{
grid-area: sidebar;
}
.item4{
grid-area: footer;
}

/* here, `.` represents the empty space */
```



▼ grid-template

- It is the shorthand for `grid-template-rows` , `grid-template-columns` , `grid-template-areas` in a single declarations.

```
.container {
  grid-template: none | <grid-template-rows> / <grid-template-columns>;
}
```

- here, we write the `grid-template-rows` values and `grid-template-columns` values together.

▼ Gaps

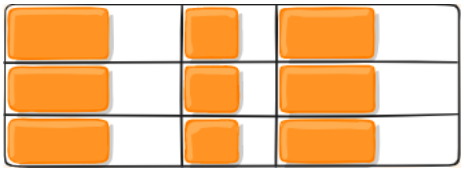
- **row-gap: s**pecifies the gaps between the rows.
- **column-gap:** specifies the gaps between the columns.
- **grid-gap:** is the shorthand for the `row-gap` and `column-gap` . this is applied to both row and column.

▼ Justify-Items

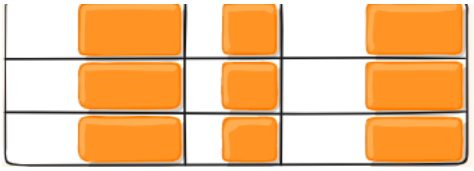
- `start` – aligns items to be flush with the start edge of their cell
- `end` – aligns items to be flush with the end edge of their cell
- `center` – aligns items in the center of their cell

```
.container {
  justify-items: start | end | center | stretch;
}
```

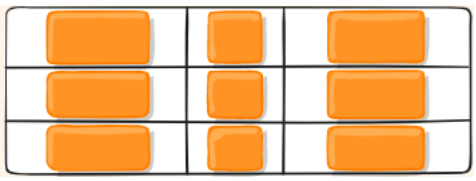
- **start:**



- **end:**



- **center:**



- **stretch:**

▼ align-Items

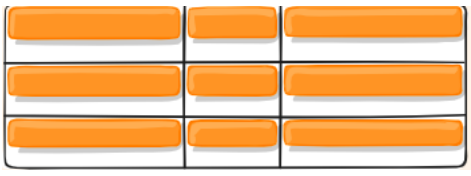
- works for row.

Values:

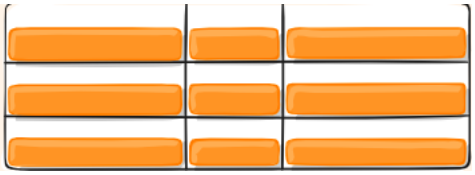
- `stretch` – fills the whole height of the cell (this is the default)
- `start` – aligns items to be flush with the start edge of their cell
- `end` – aligns items to be flush with the end edge of their cell
- `center` – aligns items in the center of their cell

```
.container {
  align-items: start | end | center | stretch;
}
```

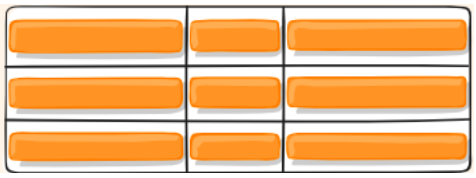
- **Start:**



- **End:**



- **Center:**

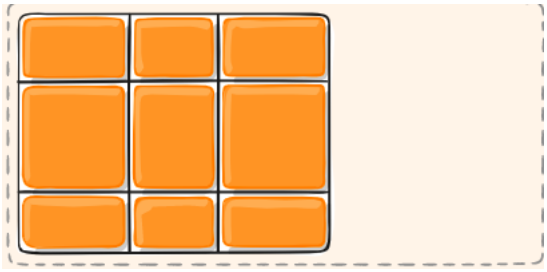


▼ Justify-content

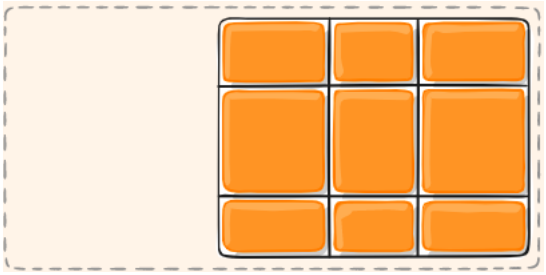
- applies to whole content

```
.container {
  justify-content: start | end | center | stretch | space-around | space-between | space-evenly;
}
```

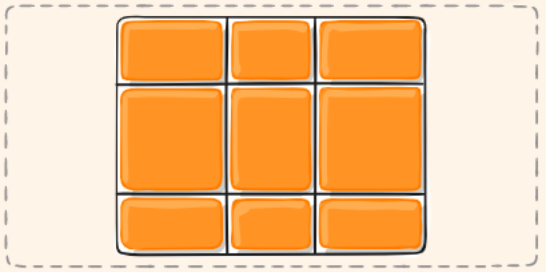
- **Start:**



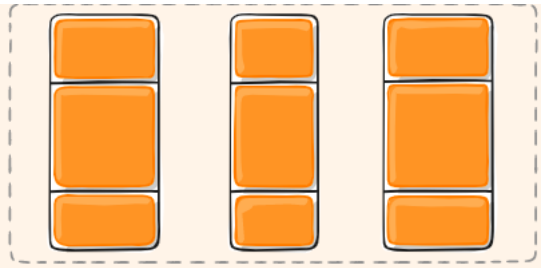
- **End:**



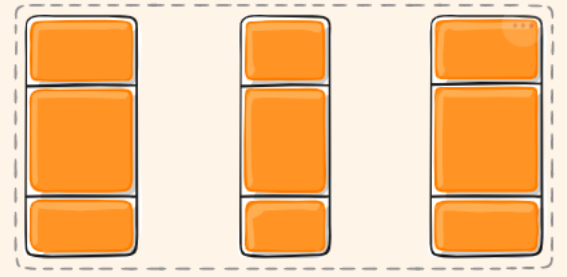
- **Center:**



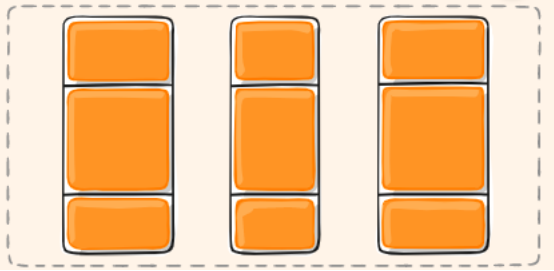
- **Space-around:**



- **Space-between:**



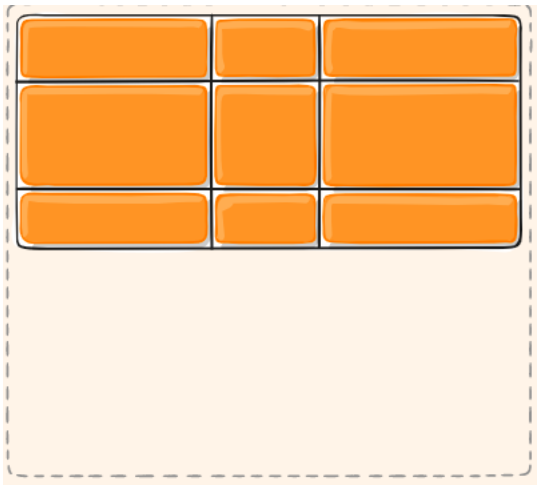
- **Space-Evenly:**



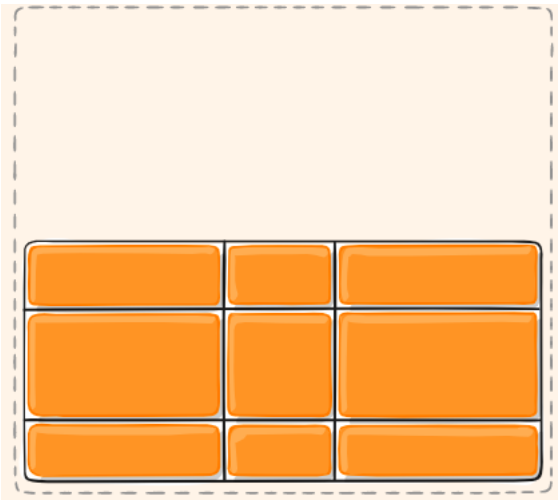
▼ Align-content

```
.container {
  align-content: start | end | center | stretch | space-around | space-between | space-evenly;
}
```

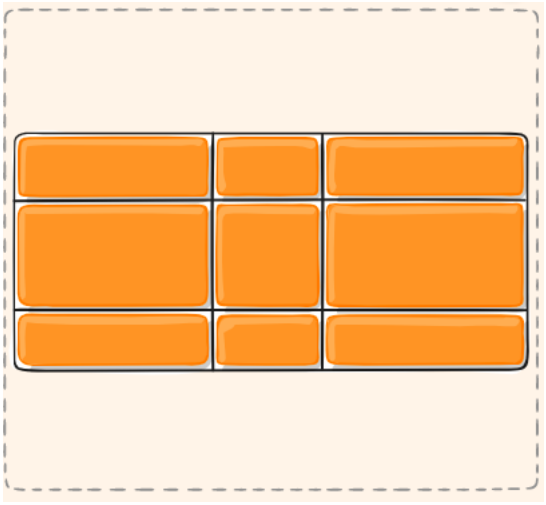
- **Start:**



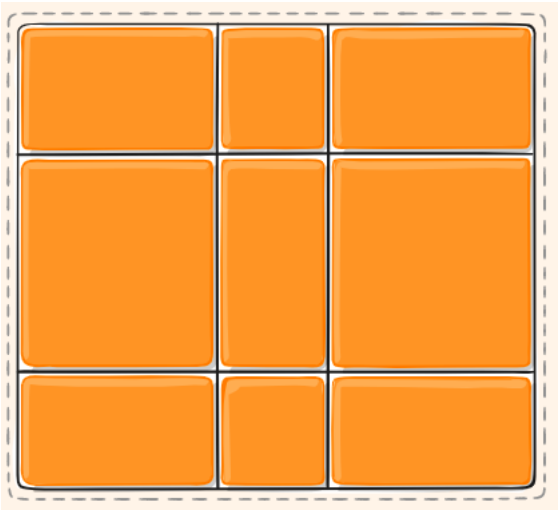
- **End:**



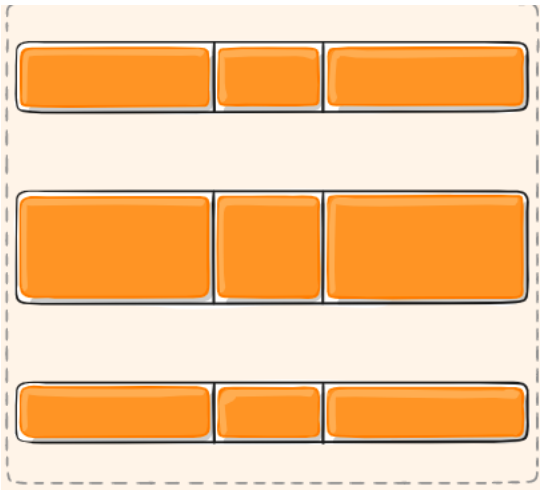
- **Center:**



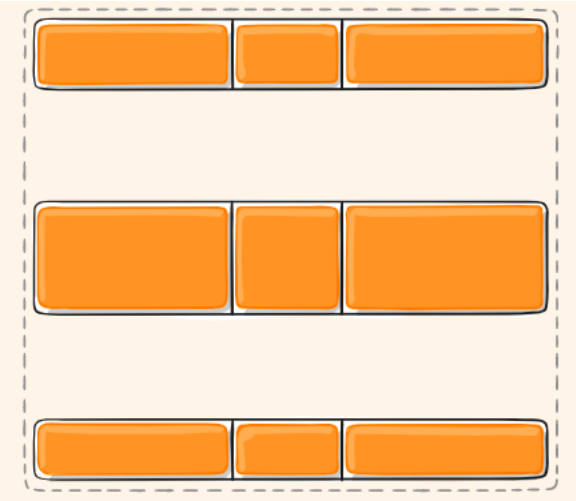
- **Stretch:**



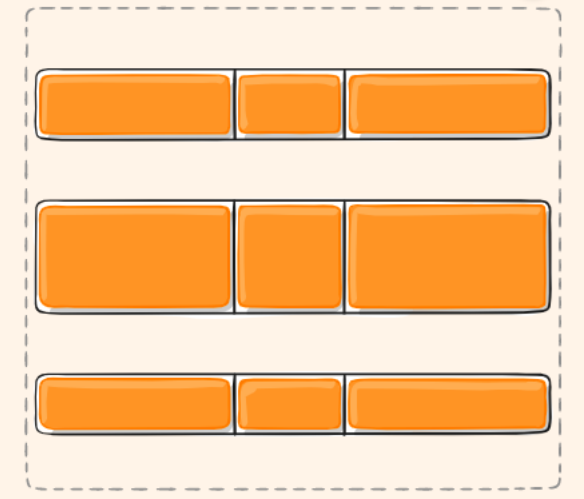
- **Space-around**



- **Space-between**



- **space-evenly**



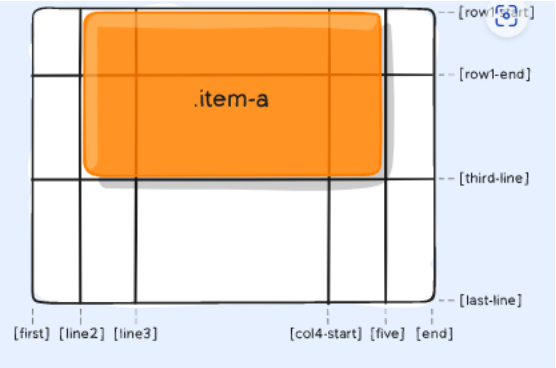
▼ Column/Row - start/end

- `grid-column-start/row-start` is the starting point from where the item begins.
- `grid-column-end/row-end` is the ending point for an item.

- **Values:** The values that is applicable are:
  - `number`
  - `name`
  - `span <number>`
  - `span <name>`
  - `auto`

```
.item-a {
  grid-column-start: 2;         // starts from 2nd column
  grid-column-end: five;        // ends at (n-1) column
  grid-row-start: row1-start;   // starts at row1
  grid-row-end: 3;              // ends at (n-1) row
}
```

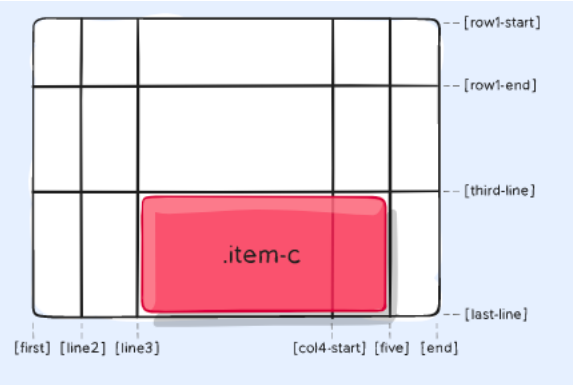
Above code can be visualized as:



▼ Grid-column/row

- it is the shorthand for `grid-column-start` and `grid-column-end` together and also for `grid-row-start` and `grid-row-end` .
- **values:**
  - `<start-line>` / `<end-line>`

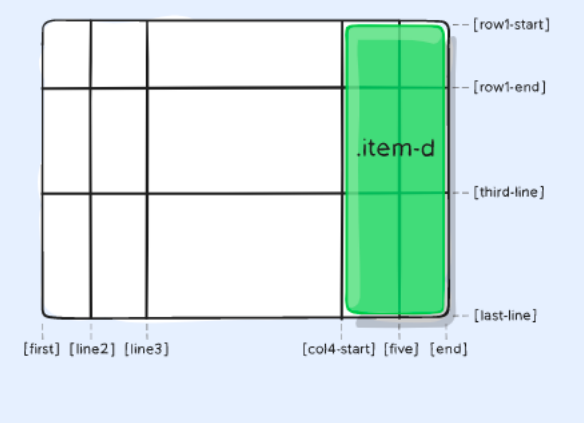
```
.item-c {
  grid-column: 3 / span 2;         // span 2 means visit 2 columns from 3rd column
  grid-row: third-line / 4;
}
```



▼ Grid-area

- it is shorthand for `grid-column` and `grid-row` combined together.
- `<row-start>` / `<column-start>` / `<row-end>` / `<column-end>`

```
.item-d {
  grid-area: 1 / col4-start / last-line / 6;
}
```



▼ Special Units and fractions

▼ fr units

- It means the portion of the remaining space

```
grid-template-columns: 1fr 3fr;
```

here, it means the two elements are divided into 25% and 75% of the total width respectively.

▼ Sizing Content

- when sizing rows and columns, we ca use px, rem, %, etc. but there are also keywords like:
- `min-content` **:** It is the minimum size of the content.
- `max-content`: max size of the content.
- `auto` :
- `fr` : fractional unit

▼ Sizing Function

- ***`fit-content()`*****:** It uses the space available. It doesn't utilize space less than `min-content` or more space than `max-content` .

- ***`minmax()`***: It sets minimum and maximum value for value what the length is able to be.

```
grid-template-columns: minmax(100px, 1fr) 3fr;
```

- ***repeat()***: The `repeat()` function can be used to repeat any section of our track listing. For example we can repeat a pattern of tracks.

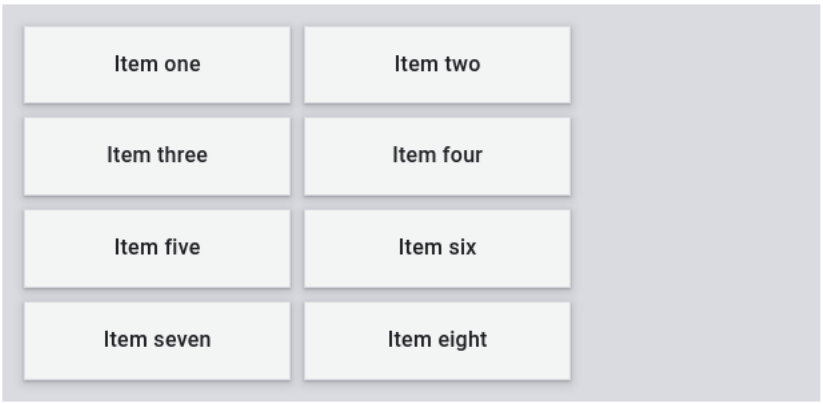
```
.container {
    display: grid;
    grid-template-columns: repeat(12, minmax(0,1fr));
}
```

- ***auto-fit and auto-fill*** : `auto-fill` fits as many possible columns as possible on a row, even if they are empty.

  `auto-fit` fits whatever columns there are into the space. It expands columns to fill space rather than empty columns.

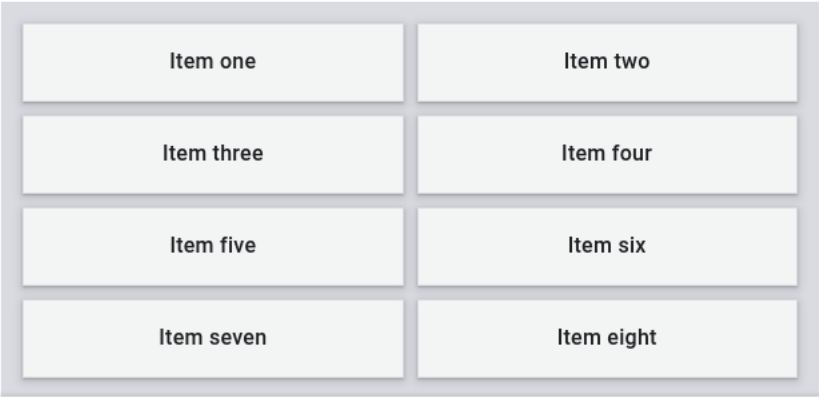
```
grid-template-columns:
  repeat(auto-fit, minmax(250px, 1fr));
```

```
grid-template-columns: repeat(auto-fill, 200px);
```




```
grid-template-columns: repeat(auto-fill, minmax(200px,1fr));
```

Resize the editor panel to see the grid change.



▼ Flow Layouts

- If not using flex-box or grid, our element display in normal flow.

▼ Inline Block

- Using inline-block we can make surrounding element respect block margin and padding.

```
p span {
  display: inline-block;
margin-top: 0.5rem;
}
```

▼ Floats

- The `float` property instructs an element to “float” to the direction specified. We can instruct element to float `left`, `right` or `inherit`.

```
img {
  float: left;
  margin-right: 1em;
}
```

▼ Multi-column Layout

- If we have long list of elements, we can use multi-column to split into multiple columns.

```
.countries {
  column-count: 2;
  column-gap: 1em;
}
```

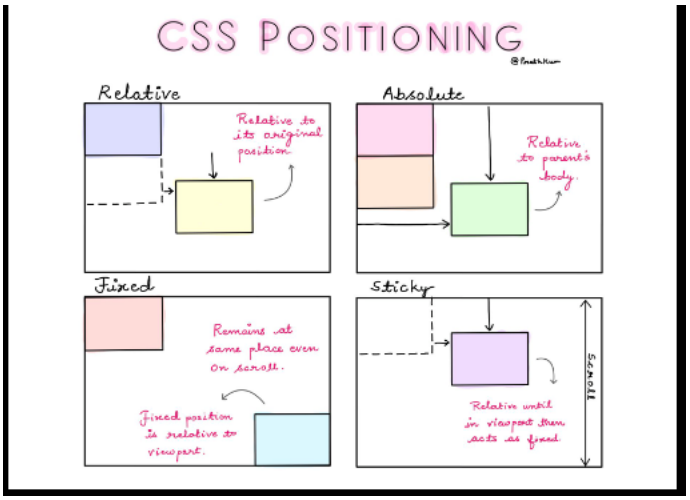
It splits the list into two columns.

```
.countries {
  width: 100%;
  column-width: 260px;
  column-gap: 1em;
}
```

Instead of setting number of columns we can define minimum desired width using `column-width.` more column will automatically be created as space is reduced, columns will also reduce.

▼ Positioning

The `position` property changes how an element behaves in the normal flow of the document, and how it relates to other elements. The available options are `relative`, `absolute`, `fixed` and `sticky` with the default value being `static`.




```
.my-element {
  position: relative;
  top: 10px;
}
```

Adding `position: relative` to an element also makes it the containing block of any child elements with `position: absolute`. This means that its child will now be repositioned to this particular element, instead of the topmost relative parent, when it has an absolute position applied to it.

- **Relative:** When an element is positioned relatively, it is positioned relative to its normal position in the document flow. You can use the top, bottom, left, and right properties to move the element from its original position. Other elements on the page will not be affected by the movement of the relatively positioned element.
- **Absolute:** When an element is positioned absolutely, it is positioned relative to the nearest positioned ancestor element (i.e., an element with a position property of relative, absolute, or fixed). If there is no positioned ancestor element, then the absolute position is calculated relative to the body element. Absolute positioning can be used to create layered layouts and pop-up menus.
- **Fixed:** The element remains fixed in the same position.
- **Static:** This is the default position of all elements on a webpage. When an element is positioned statically, it will appear in the normal flow of the document, with no special positioning applied.

▼ 4. Cascade

- Whenever two or more competing CSS rule apply to an element. There is a conflict which one to select.
- The cascade is the algorithm for solving conflicts where multiple CSS rules apply to an HTML element.

Understanding cascading helps us to know how the browser resolves conflicts like this. There are 4 distinct stages of cascade algorithm.

1. **Position and order of appearance:** determines the order in which CSS rules appear
2. **Specificity:** determines which CSS selector has the strongest match
3. **Origin:** determines the order of when CSS appears and where it comes from.

4. **Importance:** determines which CSS rule has more weight than other, or has `!important` rule assigned.

▼ Position and order of appearance

- The order in which our CSS rule appear and how they appear is taken into consideration by the cascade while it calculates conflict resolutions.

```
button {
  color: red;
}

button {
  color: blue;
}
```

Here the second CSS property is applied to the button.

- Similarly, Styles can come from various sources on HTML page, `<link>` tag and embedded `<style>` and `inline` CSS.

  The CSS is applied to the element that that has the most specific CSS applied to it.

  If the `link` is below the `style` tag the CSS inside the link tag is applied to the element.

  `inline` CSS overrides all the other CSS unless the `!important` is defined.

```
.button{
color: red;
color: green;
}
```

Here, the color green is applied to the button as it is defined last.

- If we want to specify two values for same property for multiple browsers, where one property works of the other doesn't, we can do that. This approach of declaring the same property twice works because browsers ignore values they don't understand.

▼ Specificity

- It is the algorithm to determine which CSS selector is more specific.
- CSS targeting a class on an element will make that rule more specific.

```
<h1 class="my-element">Heading</h1>
```

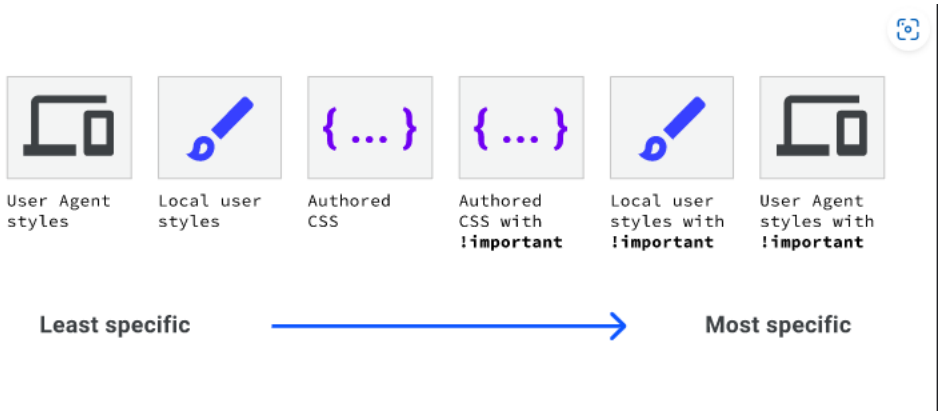
```
.my-element {
  color: red;
}

h1 {
  color: blue;
}
```

Here, in above example, the h1 will be red even though it comes after the class.

- Id makes CSS more specific, so if style is applied to an ID, it will override those applied many other ways.
- Styling component with an Id is not a good practice.

▼ Origin

- origin includes the browser's internal style-sheet, styles added by browser extensions or the operating system, and our authored CSS. The **order of specificity of these origins**, from least specific, to most specific as:
  1. **User agent base styles:** These are the styles our browser applies to HTML element by default.
  2. **Local user styles:** Can come from OS, such as base font size. They can come from browser extensions, such as browser extension that allows user to write own custom CSS for a page.
  3. **Authored CSS**: The CSS we author.
  4. **Authored** `!important`: any `!important` that we add.
  5. **Local user styles** `!important` **:** any `!important` that comes from os or extensions.
  6. **User Agent** `!important` **:** `!important` that are defined in the default CSS, provided by the browser.



▼ Importance

- The order of importance are as follows:
  1. normal rule types, such as `font-size` , `color` , etc.
  2. `animation` rule types.
  3. `!important` rule type
  4. `transition` rule type.

- Active animation and transition rule types have higher importance than normal rules. In the case of transitions higher importance than `!important` rule types. This is because when an animation or transition becomes active, its expected behavior is to change visual state.

▼ 5. Specificity

- It is the algorithm to determine which CSS selector is more specific.

  If multiple style blocks have different selectors that configure the same property with different values and target the same element, specificity decides the property value that gets applied to the element. Specificity is basically a measure of how specific a selector's selection will be:

  - An element selector is less specific and has less weight.
  - A class selector is more specific and has more weight.

```
.main-heading {
    color: red;
}

h1 {
    color: blue;
}
```

for above example class is more specific than the element tag. so output will be:

▼ Specificity scoring

- Each selector rule gets a scoring.

▼ Scoring each selector types:

a. **Universal Selector:** a universal selector `*` has no specificity and has `0 points` . Any rule with 1 or more points will override it.

```
*{
background-color: green;
}
```

b. **Element or pseudo-element selector:** A element or pseudo-element gets `1 points`.

```
// type selector

div{
color: red;
}

// pseudo-element selector
::selection{
color: red;
}
```

c. **Class, pseudo-class and attribute selector:** They get `10-points` of specificity.

```
// class selector
.className{
color:red;
}

// pseudo-class selector
:hover{
color: red;
}

// Attribute selector
[href= '#'] {
color: red;
}
```

But if `:not(_)` is used:

```
div:not(.my-class) {
  color: red;
}
```

Here, the specificity becomes `11` where `div` has `1` and `class` inside has `10` specificity.

d. **ID selector**: Id selector has `100-points` as long as `#id` is used and not attribute selector `[id="id"]` is selected.

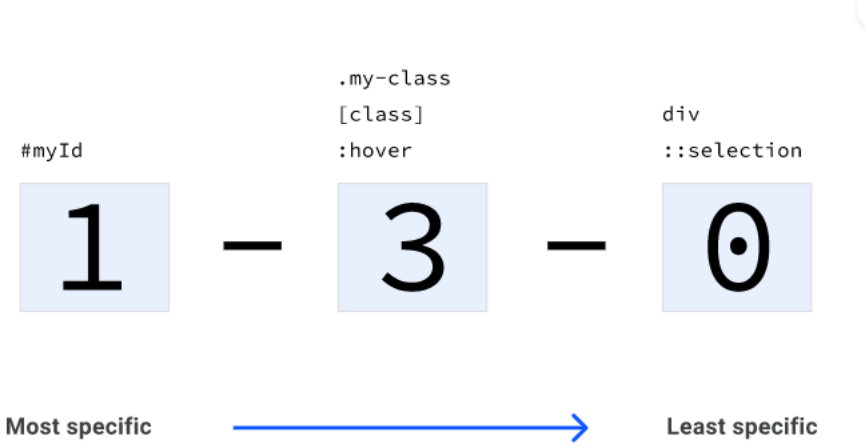
```
// id selector
#id{
color: red;
}
```

e. **Inline style attribute:** It has specificity score of `1000-point` **.**

```
// inline style attribite
<div style= "color: red"> </div>
```

f. **!Important rule:** IT has `10000-points` . This is the highest specificity that one item can get.

```
//important
.class{
color: red !important;       /* 10000 points */
color: red; /* 10 points */
}
```



▼ Specificity in context

- The specificity of each selector that matches an element is added together.

```
<a class="my-class another-class" href="#">A link</a>
```

For the above html we can write CSS as:

```
/* the a tag alone has 1-points*/
a {
  color: red;
}

/* here the specificity is 11-points as a has 1 points and class has 10 points */
a.my-class {
  color: green;
}

/*  here the specificity is 21*/
a.my-class.another-class {
  color: rebeccapurple;
}

/* specificity = 31 points */
/*  href also has 10 points*/
a.my-class.another-class[href] {
  color: goldenrod;
}

/* specificity = 41 points */
/*  :hover also has 10 points*/
a.my-class.another-class[href]:hover {
  color: goldenrod;
}
```

▼ 6. Sizing Units

Numbers:

- Numbers are usually used to define `opacity`, `line-height` and for color values `rbga` , etc.

```
p{
opacity: 0.5;
line-height: 2;      // if p has font size 12px line height here will be 24px
color: rgb(1,2,3);
transform: scale(1.2)  // item is scaled to 120%.
}
```

Percentages:

- percentage is relative to its parent or the size of view port.

```
.box{
width: 50%;
margin: 50%;
}
```

▼ Dimensions and lengths:

- If we attach unit to a number, it becomes the dimension.

Absolute Lengths:

- The length remains `consistent` all over.

Unit	Name	Equivalent to
cm	Centimeters	1cm = 37.8px
mm	Millimeters	1mm = 1/10th of 1cm
Q	Quarter-millimeters	1Q = 1/40th of 1cm
in	Inches	1in = 2.54cm = 96px
pc	Picas	1pc = 1/6th of 1in
pt	Points	1pt = 1/72th of 1in
px	Pixels	1px = 1/96th of 1in

Relative lengths:

- Relative length units are relative to something else, perhaps the size of the parent element's font, or the size of the viewport.

Unit	Relative to
`em`	Relative to the font size, i.e. 1.5em will be 50% larger than the base computed font size of its parent.
`ex`	Heuristic to determine whether to use the x-height, a letter "x", or `.5em` in the current computed font size of the element.
`ch`	The advance measure (width) of the glyph "0" of the element's font.
`rem`	Font size of the root element (default is 16).
`lh`	Line height of the element.
`rlh`	Line height of the root element. When used on the `font-size` or `line-height` properties of the root element, it refers to the properties' initial value.
`vw`	1% of the viewport's width.
`vh`	1% of the viewport's height.
`vmin`	1% of the viewport's smaller dimension.
`vmax`	1% of the viewport's larger dimension.
`vb`	

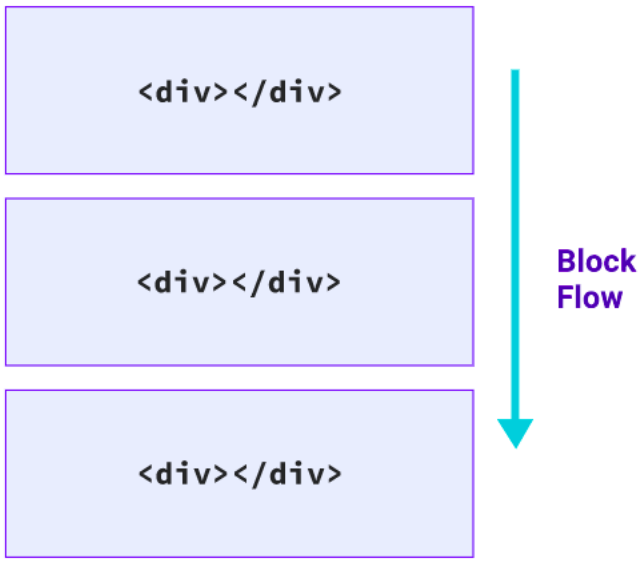
	1% of the size of the initial containing block in the direction of the root element's block axis.
`vi`	1% of the size of the initial containing block in the direction of the root element's inline axis.
`svw, svh`	1% of the small viewport's width and height, respectively.
`lvw, lvh`	1% of the large viewport's width and height, respectively.
`dvw, dvh`	1% of the dynamic viewport's width and height, respectively.

▼ 7. Logical Properties

- Logical properties in CSS are a way of describing the spacing, alignment, and other visual aspects of elements on a webpage that takes into account the direction in which content flows, like text or images.
- The physical properties of top, bottom, right and left refers to the physical dimensions of the viewport.
- Logical properties refers to the edges of a box as they relate to the flow of content. and can change if the text direction or writing mode changes.

Block Flow:

- It is the direction in which content blocks are placed. If there are two paragraph block flow is where ta second paragraph will go. Example: a paragraph follows top-bottom flow.



Inline Flow:

- It is how the text flows in a sentence, In English the inline flow in left to right.

- flow is determined by document's writing mode.

```
writing-mode: vertical-lr | vertical-rl;
/*read line from vertial letft-right or right-left */
```

Flow relative:

- similar to `margin-top` property for physical properties, we can use `margin-block-start` in logical properties.

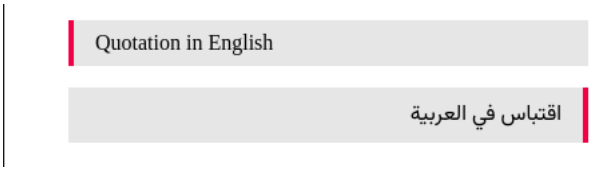
▼ Logical Properties compared to physical properties

1. Sizing:

Physical property	Logical property
`width`	`inline-size`
`max-width`	`max-inline-size`
`min-width`	`min-inline-size`
`height`	`block-size`
`max-height`	`max-block-size`
`min-height`	`min-block-size`

2. Borders:

Physical property	Logical property
border-top	border-block-start
border-bottom	border-block-end
border-left	border-inline-start
border-right	border-inline-end
border-top-color	border-block-start-color
border-top-width	border-block-start-width
border-top-style	border-block-start-style
border-top and border-bottom	border-block
border-left and border-right	border-inline



3. Margin:

Physical property	Logical property
margin-top and margin-bottom	margin-block
margin-left and margin-right	margin-inline

4. Padding:

Physical property	Logical property
padding-top	padding-block-start
padding-bottom	padding-block-end
padding-left	padding-inline-start
padding-right	padding-inline-end
padding-top and padding-bottom	padding-block
padding-left and padding-right	padding-inline

5. Positioning:

Physical property	Logical property
top	inset-block-start
bottom	inset-block-end
left	inset-inline-start
right	inset-inline-end
top and bottom	inset-block
left and right	inset-inline

6. Text-alignment:

```
// for physical properties
text-align: left | end| etc. ;
```

Physical value	Writing mode	Equivalent to:
`start`	LTR	`left`
`start`	RTL	`right`
`end`	LTR	`right`
`end`	RTL	`left`

7. Border Radius:

Physical property	Logical property
`border-top-left-radius`	`border-start-start-radius`
`border-top-right-radius`	`border-start-end-radius`
`border-bottom-left-radius`	`border-end-start-radius`
`border-bottom-right-radius`	`border-end-end-radius`

8. etc.

▼ 8. Z-index and stacking context

- `z-index` specifies the stack order of the element.
- An element with greater stack order is always in front of an element with a lower stack order.
- If two positioned elements overlap without a `z-index` specified, the element positioned last in the HTML code will be shown on top.
- we need to set the element's `position` to other than `static` to see z-index.

- But for flex or grid, we don't need to add `position: relative`

▼ 9. Animations

▼ Keyframe

- It is the mechanism that is used to assign animation states to timestamps, along a timeline.

```
@keyframe custom_animation_name {     // it is case sensitive
from {
}
to{
}
}
```

- The custom identifier works as the function name. we can use this keyframe anywhere in the module.
- The `from` and `to` are the keywords that represents `0%` and `100%`, which are the start and the end of the animation

```
@keyframe custom_animation_name {     // it is case sensitive
0% {
}
100%{        // can be any
}
}
```

▼ Animation properties

1. animation-duration

```
.element{
animation-duration: 10s;
}
```

- It defines how long the `@keyframes` timeline should be. It is the time value.
- It's default value is 0.
- We cannot add negative value to the animation duration

2. animation-timing-function

- To create natural motion in animation
- Animation runs at variable speed over the course of animation duration.
- If the value is calculated beyond that of value defined in `@keyframe` , it makes the element to bounce.
- `linear` , `ease`, `ease-in`, `ease-out` , `ease-in-out` are some of the properties of animation timing function.

```
.element{
animation-timing-function: ease-in-out;
}
```

Steps easing function:

- create intervals for the animation to move regardless of the curve.

- `steps()` function lets us break the timeline into defined, equal intervals.

```
.element{
animation-timing-function: steps(10, end);
}
/* first argument is the steps, if steps are 10, there are 10 keyframes,
  each keyframe will play in sequence for the exact amount of time.

second argument is the direction. if its `end` (default), steps finishes
at the end of the timeline. If set to `start`. the first step of our animation
completes as soon as it starts, which means it ends one step earlier than `end`

 */
```

3. animation-iteration-count

- It defines how many times the `@keyframe` timeline should run. By default it is `1` , means whenever the animation reaches the end of our timeline, it stops running.
- The number cannot be negative.

```
,animate{
animation-iteration-count: 5;
}
```

- we can use `infinity` to loop our animation.

4. animation-direction

```
.my-element {
  animation-direction: reverse;
}
```

- We can set the direction of our animation .
- `normal`: default value, which is forward
- `reverse`: it runs backwards over our timeline
- `alternate`: timeline will run forward or backward in a sequence.
- `alternate-reverse`: reverse of alternate

5. animation-delay

```
.eanimation{
animation-delay: 5s;
}
```

- It defines how long to wait before starting the animation.
- It accepts the time value.

6. animation-play-state

```
.animate{
animation-play-state: paused;
}
```

- It allows us to play or pause the animation.
- The ***default value*** is `running` and if `pause` is set it pause the animation.

7. Animation Shorthand:

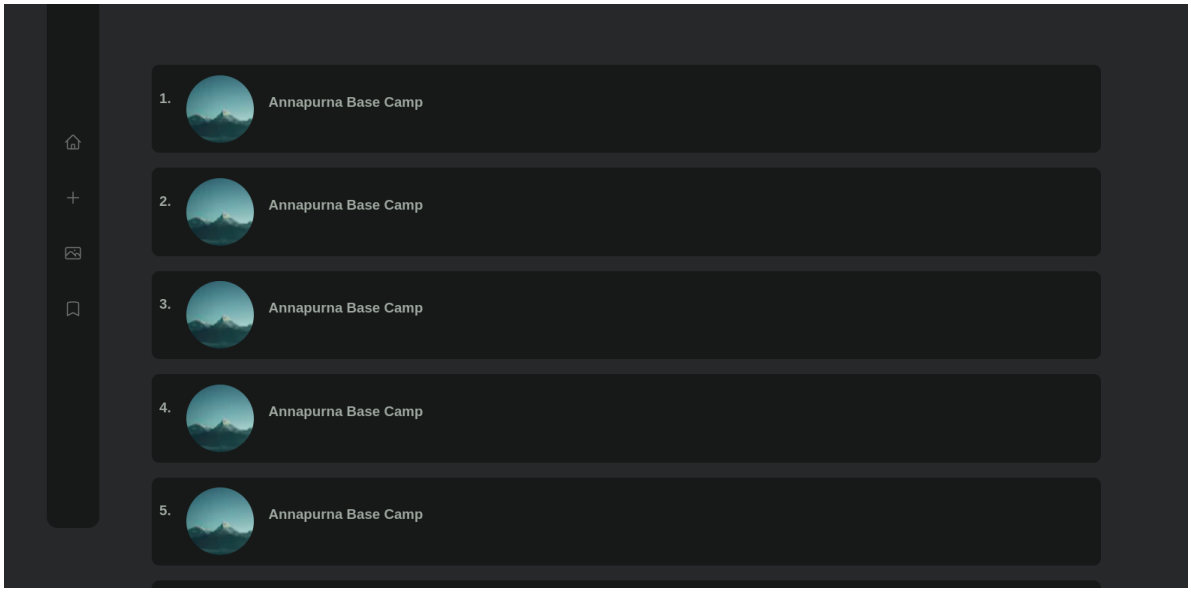
```
.element{
animation: animation_name animation_duration animation_timing_function
 animation_delay aniumation-iteration-count animation_direction
animation_fill_mode animatipon_play_state
}
```

- Instead of defining the animation properties seperately, we can define them in single `animation` shorthand.

▼ 10. Overflow

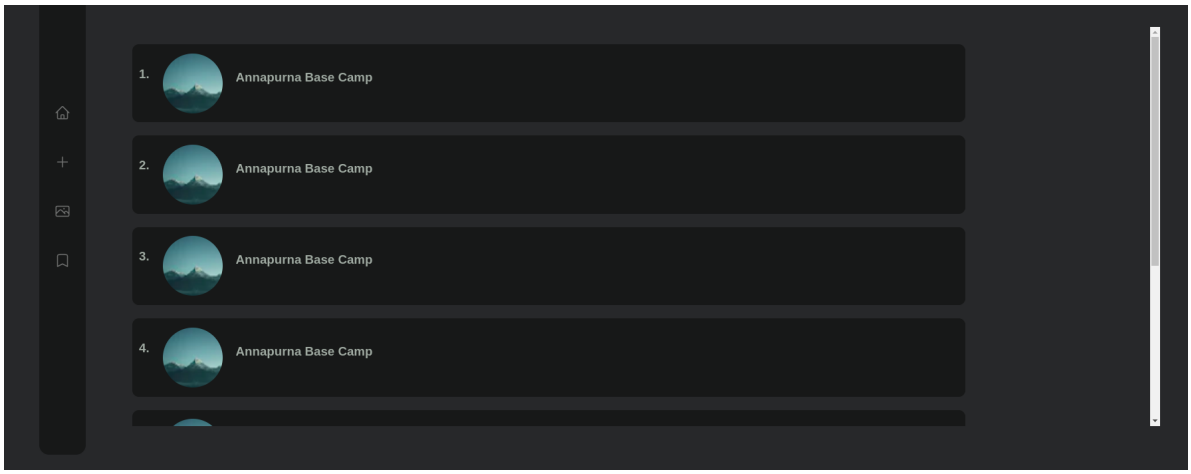
- It is the way to deal with the content that doesn't fit in,
- When content extends beyond its parent, we can use overflow property.
- Scrollbars in `overlay` mode will overlap the padding, and when in `inline` mode will add to the padding.

- Normally it may look this:



```css
.list__items {
  display: flex;
  flex-direction: column;
  /* overflow-y: scroll; */
  height: 600px;
  /* width: 100%; */
}
```

- But when overflow property is used we can have:



OPTIONS

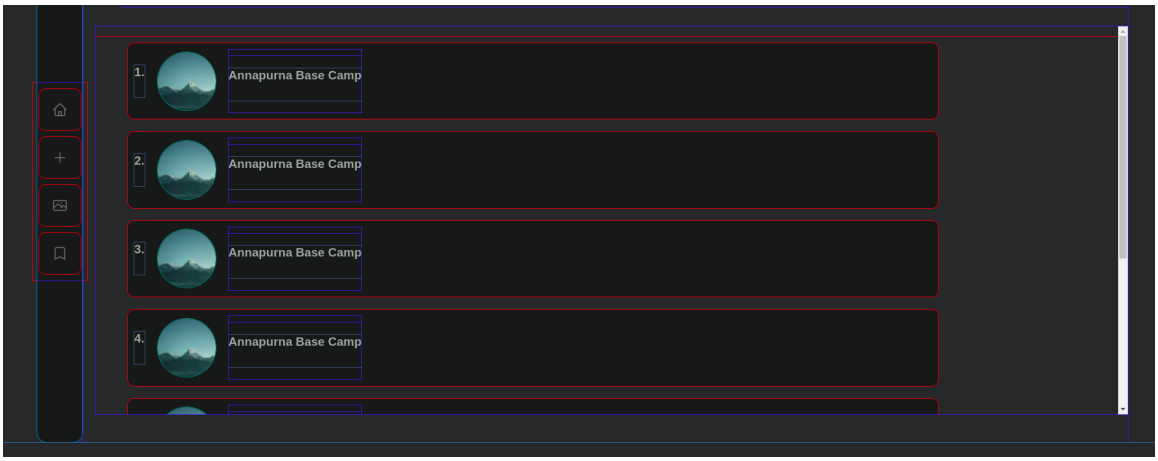
1. text-overflow:

- **I**t helps with the control of individual line of text.
- It specifies how the text will appear when it doesn't fit in the available space of the element.
- To use `text-overflow` we need single unwrapped text, and `overflow:hidden` and `white-space` value that prevent wrapping.

```
p{
texti-overflow: elipsis | clip;
overflow: hidden;
white-space: nowrap;
}
```

▼ 2. Overflow:

- Overflow is set on an element to control what happens when its children needs more space than it has available.
- Same example as above images. when overflow is set content becomes scrollable in the defined axis and fits inside the parent's div.



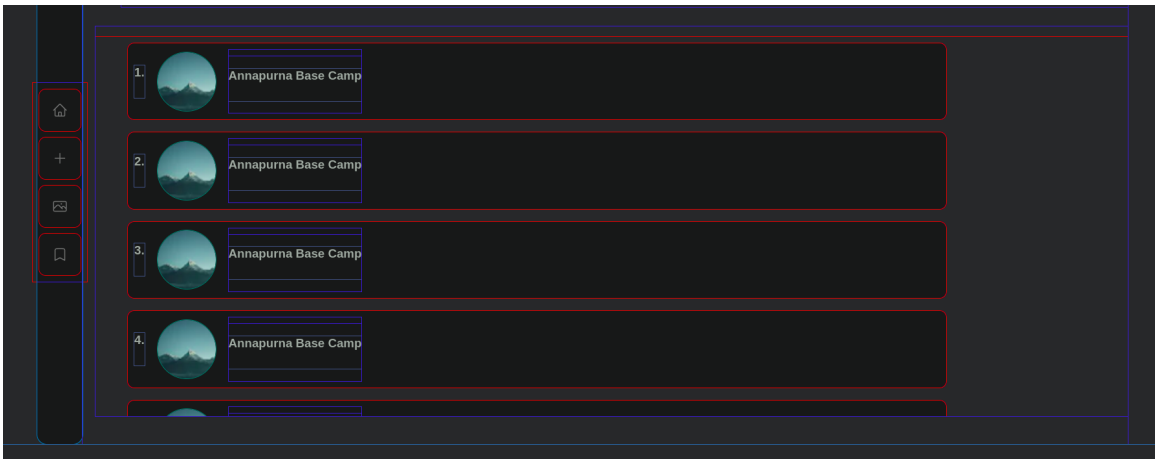
a. `overflow-x`: it is the property that controls overflow along the x -axis.

b. `overflow-y`: it is the property that controls overflow along the y-axis.

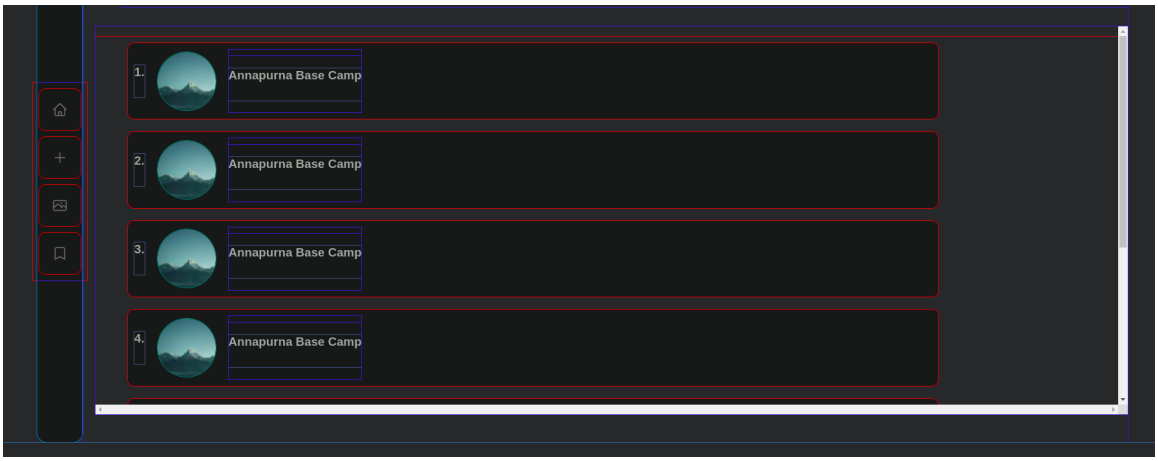
```
// SHORTHAND FOR OVERFLOW

p{
overflow: hidden scroll;
}

/* overflow: overflow-x overflow-y; */
// if one keyword specified, it applies to both the axes.
```

Values:

1. `overflow: visible` **(default)** : content is not hidden.
2. `overflow:hidden`: no overflow only content that fits inside the parent box is visible. no scrollbar is present.



3. `overflow:scroll`: we get a scrollbar even though the content is not overflowing.



4. `overflow:clip`: similar to `overflow-hidden` . It forbids any kinds of scrolling.
5. `overflow:auto`: This respects the user's preferences and shows scrollbars if needed, but hides them by default.

▼ 11. Blend Modes

- We can blend background-images together or blend them with background color.

```
.blend_example{
background-image: url();
background-color: cyan;
background-blend-mode: multiply;
```

Background Blend Mode

- We can use `background-blend-mode` when we have multiple backgrounds on an element.

Mix Blend Mode

- It affects the entire element.

▼ Modes

- `normal`: This is default blend mode and changes nothing about how an element blends with other.
  - `Multiply` : Stacking multiple transparencies on top of each other. White pixel appears transparent and black pixels appears black. anything in between will multiply its `light` values. This means lights get much lighter and darks, darker - most often producing darker result. `white becomes transparent`, `black becomes black`, `anything in between becomes light` .

```
mix-blend-mode: multiply;
```

▼ 12. Gradient

- To have multiple shades of color together
- gradient allows to create anything from smooth gradient between two colors.

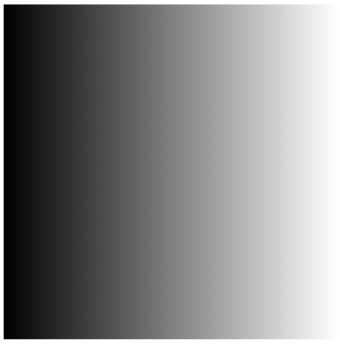
▼ Linear Gradient

- It generates image of two or more colors, splits them evenly while blending them.

```
.my-element {
  background: linear-gradient(black, white);
}
```

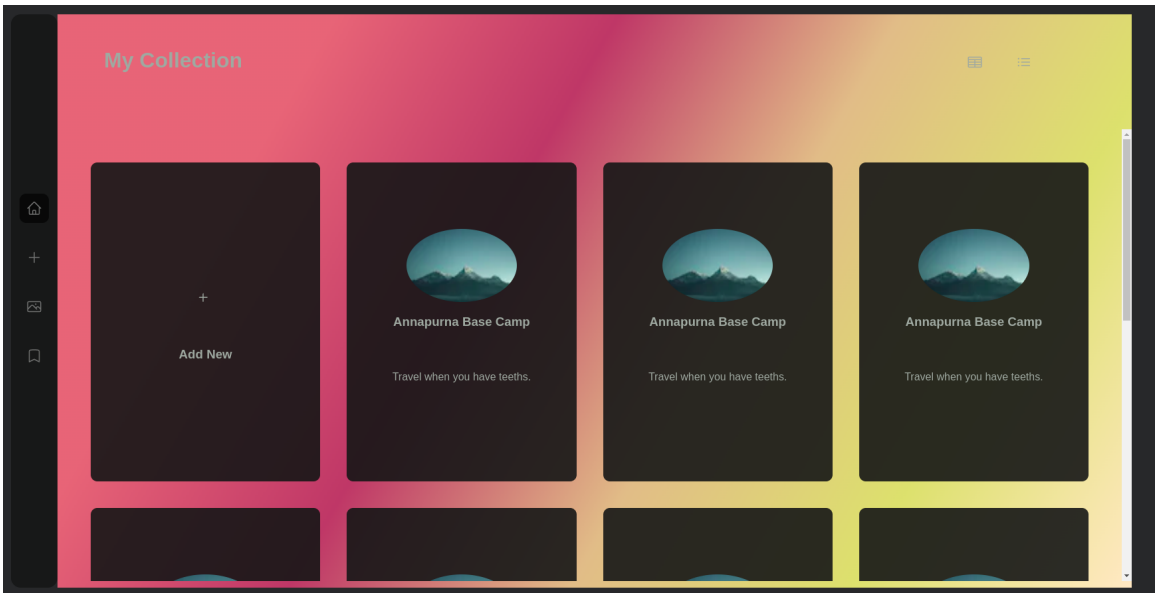
- We can specify angles after `to` keywords.
- If we want a gradient black and white that runs from black to white. we specify `to right` as the first argument.

```
.my-element {
  background: linear-gradient(to right, black, white);
}
```



- We can also specify the angle for a color like

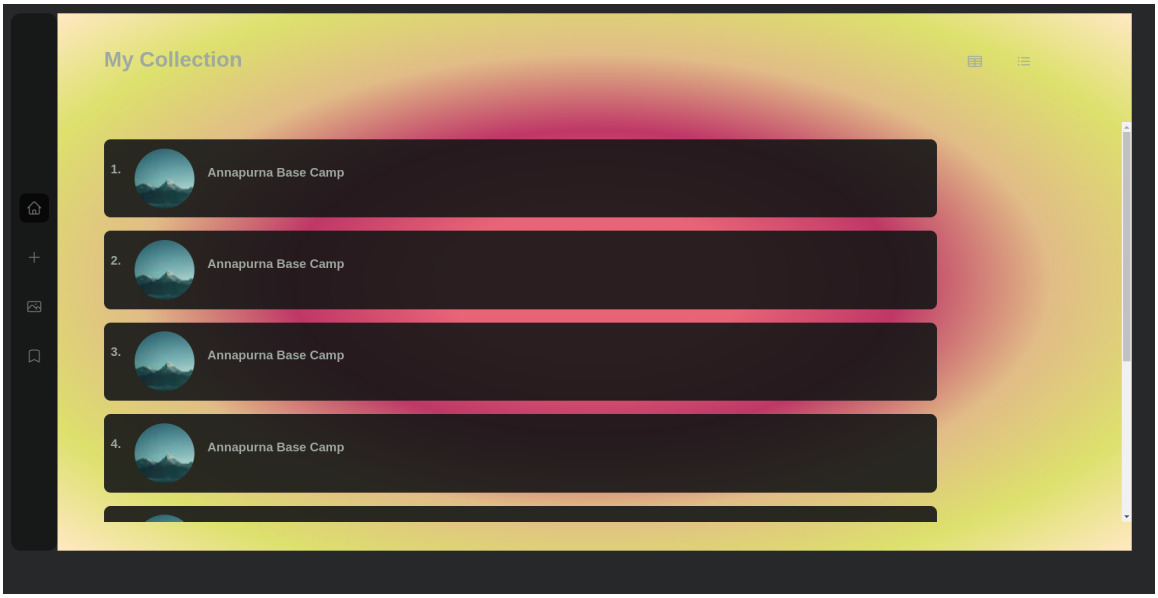
```
.my-element {
  background: linear-gradient(120deg, darkred 20%, crimson, darkorange 60%, gold, bisque);
}
```



▼ Radial Gradient

- to generate gradient in circular fashion
- unlike specifying angle in linear gradient, er specify position and ending shape.
- If not positioned, the auto position specified is `center` and selects either circle or ellipse depending on the size of the box.

```
background: radial-gradient(
    rgb(219, 121, 121) 20%,
    rgb(179, 87, 105),
    rgb(223, 183, 135) 60%,
    rgb(226, 208, 106),
    bisque
  );
```

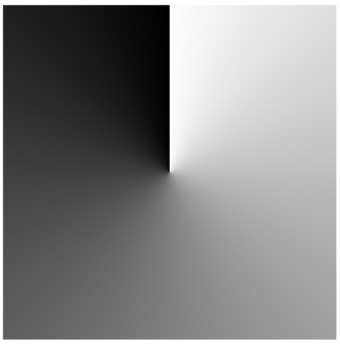


- We can specify the positions as:
  - `closest-corner`: will meet the closest corner of the center.
  - `closest-side`: will meet the side of the box closest to center
  - `farthest-side`: is opposite to `closest-side`.

▼ Conic Gradient

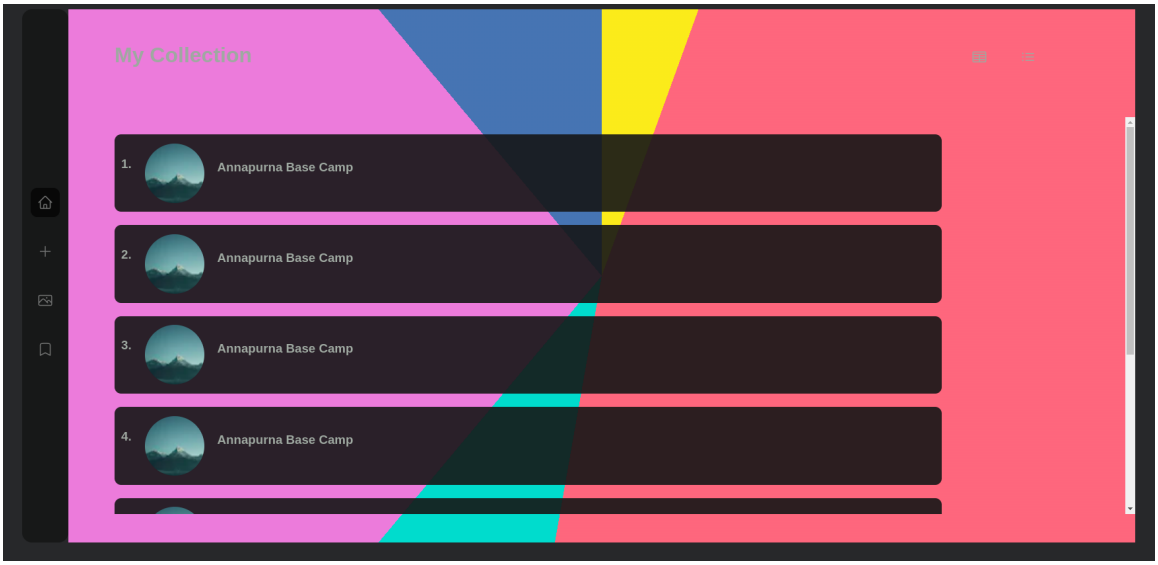
- Gradient dinxa but 360 angle ma move hunxa.

```
.my-element {
  background: conic-gradient(white, black);
}
```



- we can define positions and angles. default angle is `0`
- position is `center` by default.
- We can add as many color as we like.

```
background: conic-gradient(
     gold 20deg, lightcoral 20deg 190deg, mediumturquoise 190deg 220deg, plum 220deg 320deg, steelblue 320deg);
```



▼ Repeating gradient

- same as other gradient and tasks same argument. Difference is that if the gradient can be repeated to fill the box it will.
- `repeating-linear-gradient()`, `repeating-radial-gradient()` and `repeating-conic-gradient()`.

▼ JavaScript

▼ ES6

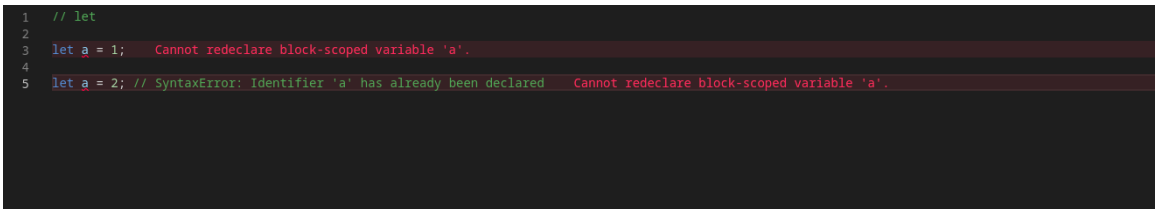
```
1. Let, const (block Scoping)
2. Arrow functions
3. this
4. Spread and rest operators
5. destructuring (object, array)
6. Promises
7. async await
8. Map and set
9. isNaN
10. Template string
11. for-of operation
```

▼ Let and const

1. `Let`

- It is block scope so it cannot be used outside the block.
- Variable declared with let cannot be re-declared.
- Variable defined with `let` must be declared before use.

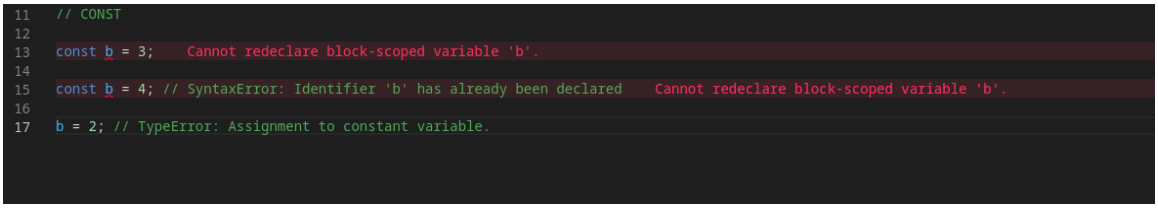
```
// let:
let a = 1;
let a = 2;    // cannot be redeclared
a = 4;       // but can be reassigned
```



2. `Const`

- variable defined with const cannot be redeclared, reassigned.
- Variables defined with const has block scope.
- unlike `let` it cannot be reassigned

```
//const
const b = 3;
const b = 6;        // cannot be redeclared
b = 19;             // cannot be reassigned as well
```



▼ Arrow Function

- It is the shorthand for writing function in javascript.
- we don't need the function keyword.
- They `aren't hoisted`. Must be defined before use.
- With arrow function there is no binding of `this` keyword.
- In regular function `this` represents the object that called the function, that can be window, document, a button or whatever. But in arrow function `this` always represent the object that defined the arrow funtion.

```
// normal function:
function function_name(args){
    // code here
}

// arrow function:
const var_name = (args) => {
    // code here
}
```

▼ Default Parameter

- It allows us to give default values to the function parameters.

```
function sum (a=2, b=4){
  return a+b;
}

sum();       // if no argument given, it takes the default values
sum(1);      // in this case the value of a is 1 and b is default value
sum(1,2);    // in this case both 1 and 2 are the new value of the parameters
sum(undefined) // sum takes the default value
```

▼ Spread and Rest Operators

Spread

- It is used to expand or spread an iterable or an array.

```
const arr = [1, 2, 3];
console.log(...arr); // 1 2 3
```

- We can `copy an array` using spread operator.

```
const arr1 = [1,2,3]
const arr2 = [...arr1, 4,5,6]
console.log(arr2);      // 1 2 3 4 5 6
```

- `cloning array`:

```
// normally
let arr1 = [1,2,3]
let arr2 = arr1;

arr1.push(4);         // change in arr1 also creates change in arr2 as it is
                      // refering to same array
console.log(arr1, arr2);
```

```
// using spread operator
let arr1 = [1,2,3]
let arr2 =[...arr1];

arr1.push(4);          // since arr2 is copy of arr1 and doesn't refer to
                       // same array there is no change in arr2
console.log(arr1, arr2);
```

- `same goes with the object`

Rest

- When spread operator is used as parameter, it is known as rest parameter.

```
const result = (...args) => {
    return args;
}
console.log(result(1, 2, 3))          // 1, 2, 3
```

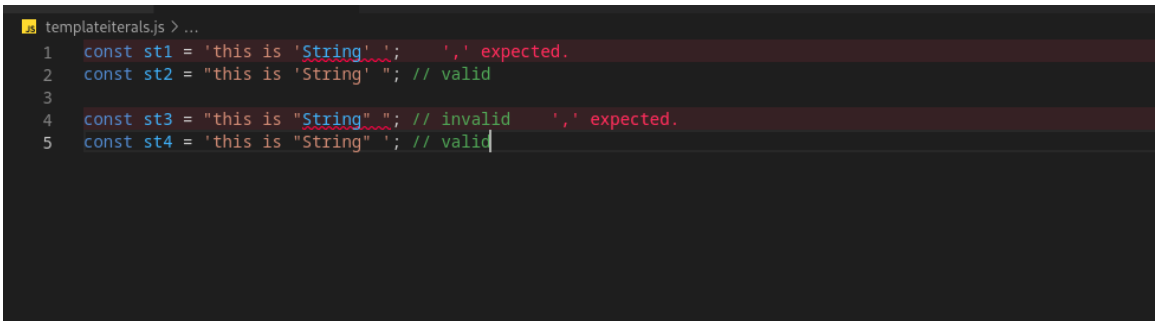
```
const res = (x,y) => {
return x+y;
}
const num = [1,2,4]
console.log(res(...num));  // function res takes first 2 parameters
```

▼ Template Literals

- similar to `f-string` in python.
- we enclose string inside `` ` ` `` .

```
const name = 'ashwin';
console.log(`hello ${name}`;       // hello ashwin
```

- In previous version:



This creates a problem. So, we use use template literal or escape characters.

```
//escape character
const st1 = 'this is \'string\' ';  // this is a valid way but doesn't look good
```

```
// instead we can use template literals
const st1 = `this is "string" `;
```

- Writing multi line string:

```
const str = ` this is line 1
this is line 2
this is line 3 `;
```

- `Expression interpolation`: process of assigning variables and expressions inside the template literal is expression interpolation.

```
//normally to use or concat variables we use + operator
const name = "abc";
console.log("hi" + name);

// but with template literal we can:
console.log(`hi ${name} `);
```

- `Tagged Template`: we can create tagged template using template literals. We use tags that allows us to parse template literals with a function.

  It is like calling a normal function, instead we don't write parenthesis `()`

```
const greet = (name) => {
console.log(`hello ${name}`;
}

greet`ashwin`;          // output: hello ashwin
```

  - `multiple argument in tagged template`:

```
const greet = (name, lname) => {
    const str1 = name;
    const str2 = lname;

    if (true) {
        console.log(`hello ${name} ${lname}`);
    }
}

const lname = ` khatiwada`;
greet`ashwin ${lname}`;
```

▼ Destructuring Assignment

- It is easy to assign array values and object properties using destructuring property.

```
// assigning object attributes to variables
const person = {
    name: 'abc',
    age: 25,
    gender: 'male'
}

// destructuring assignment
let { name, age, gender } = person;

console.log(name, age, gender);
```

- We should `use same name` for the variable as the corresponding object key.
- If we want different variable names we can:

```
let { name: name1, age: age1, gender:gender1 } = person;
```

- `Array Destructuring`:

```
const arr = [1,2,3];

const [x,y,z] = arr;
```

- `assign default value`:

```
const person = {
name: `sga`
}

const { name, age = 26} = person;
```

- `Skip item`:

```
const arr = [1,2,3]
const [x, , z] = arr;
```

- `assign remaining elements to single variable`:

```
const [x, ...y] = arr;       // ...rest cannot be the first argument
```

▼ Callback and Promises

▼ Callback

- When a `function is passed as an argument  to a function`.
- This function that is passed as an argument inside of another function is called callback function.

```
// functoin
function greet(a, fxn){
console.log(`hi ${a}`);
fxn();
}

//callback function
function callMe(){
console.log(`hey`);
}

// passing a function as an argument
greet(`Ashwin`, callMe);
```

`Callback Hell`:

- Multiple nested callbacks are used to handle async operations, making code difficult to read, understand, debug and maintain.

```
asyncFunction1(() => {
  asyncFunction2(() => {
    asyncFunction3(() => {
      asyncFunction4(() => {
        // more nested callbacks...
      });
    });
  });
});
```

- To deal with callback hell hell we can use `promises` and `async-await`.
- `promise` to deal with above problem:

```
// this helps to chain async operations in more readable and maintainable way
asyncFunction1()
  .then(asyncFunction2)
  .then(asyncFunction3)
  .then(asyncFunction4)
  .catch((err) => {
    console.error(err);
  });
```

- `async/await:`

```
// it is new syntax that allows us to write async code in a sync way
async function myFunction() {
  try {
    await asyncFunction1();
    await asyncFunction2();
    await asyncFunction3();
    await asyncFunction4();
  } catch (err) {
    console.error(err);
  }
}
```

▼ Promises

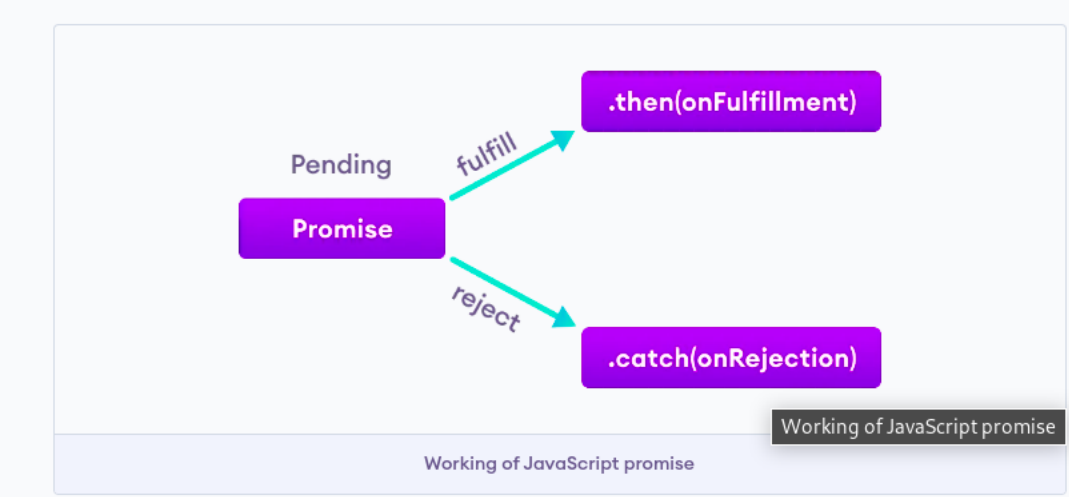
- It is a good way to handle async operations. It is a way to find out if the async operations are successfully completed or not.
- It has three state: `pending`, `fulfilled or resolved` and `rejected`
- When promise starts its in a pending state, if the process is not complete, it is in pending state.
- If the promise operation is successful. it ends in fulfilled state, else is in rejected state.

- Creating a promise:

```
let promise = new Promise(function (resolve, reject){
    // code here
}
```

- The `Promise()` constructor takes function as an argument and accepts two functions `resolve()` and `reject()`.
- If function returns successfully, the `resolve` function is called else `reject` is called.
- Example:

```
const state = true;

let stateValue = new Promise(function(resolve, reject){
if(state){
resolve("This is resolve state");
return;
}
reject(`this is reject state`;
})
console.log(state);
// output Promise {<resolved>: "There is a count value."}
```

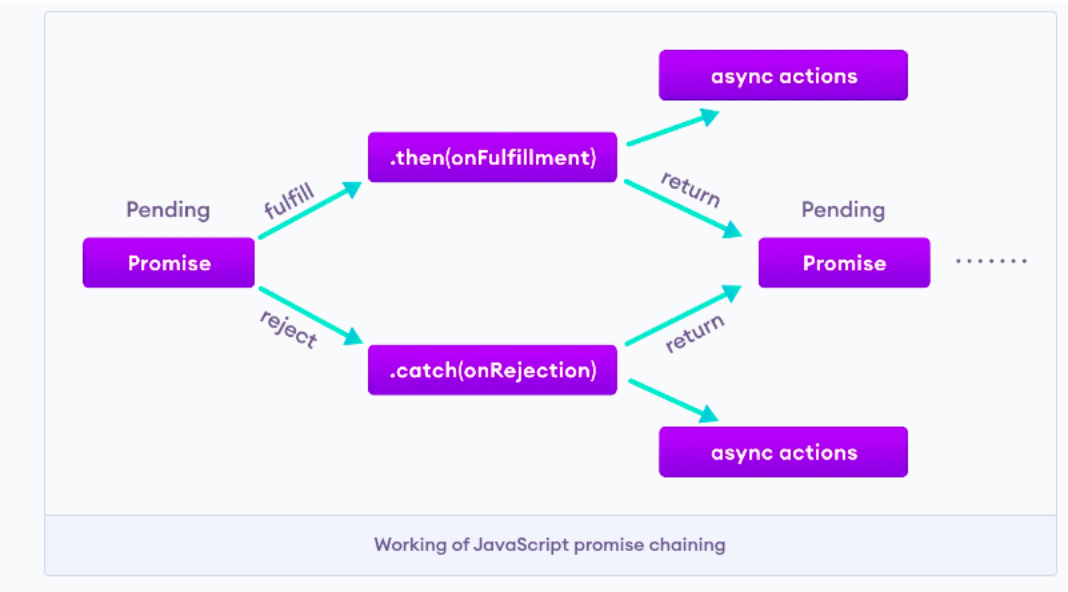


Working of JavaScript promise

`Chaining Promises`:

- the `then()` method is is used with the callback when the promise is successfully fulfilled or resolved.

```
let stateVal = new Promise(function (resolve, reject){
resolve("Promise resolved");
});

// executes when promise is resolved successfully
stateVal
.then(function successVal(result){
console.log(result);
});
.then(successVal1(){
console.log(`we can write multiple then methods`);
});
.catch((err) => {      // we can write .catch to catch errors if any
    console.error(err);
  });
```



Working of JavaScript promise chaining

`Callback vs Promises:`

- Both are used to handle `async` tasks.
- Both can be used to perform `sync` tasks as well.

Promises	Callbacks
User friendly and easy to read.	Difficult to understand.
Error handling is easy to manage.	Error handling may be hard to manage.

```
// Promise

api().then(function(result) {
    return api2() ;
}).then(function(result2) {
    return api3();
}).then(function(result3) {
    // do work
}).catch(function(error) {
    });
```

```
// Callback

api(function(result){
    api2(function(result2){
        api3(function(result3){
             // do work
            if(error) {
                // do something
            }
            else {
                // do something
            }
        });
    });
});
```

`Promise methods:`

- There are various methods available to the promise object.
  1. `all(iterable)`: waits for all promise to be resolved or any one to be rejected. If any one fails, all the process fails and doesn't looks for other rejections.
  2. `allSettled(iterable)`: waits until all promises are either resolved or rejected. Catches all the rejections.
  3. `any(iterable)`: returns the promise value as soon as any one of the promise is fulfilled or resolved.
  4. `race(iterable)`: waits until any of the promise is resolved or rejected.
  5. `reject(reason)`: returns promise object that is rejected for given reason.
  6. `resolve(value)`: returns promise object that is resolved with the given value.
  7. `catch()`: rejection handler for callback.
  8. `then()`: resolver handler for callback.
  9. `finally()`; handler for promise.
     - `Example`:

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Promise 1 resolved'), 1000);
});

const promise2 = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error('Promise 2 rejected')), 500);
});

const promise3 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Promise 3 resolved'), 1500);
});

// Promise.all()
Promise.all([promise1, promise2, promise3])
  .then((values) => {
    console.log(values); // output: [ 'Promise 1 resolved', Error: Promise 2 rejected, 'Promise 3 resolved' ]
```

```
  })
  .catch((err) => {
    console.error(err); // output: Error: Promise 2 rejected
  });

// Promise.allSettled()
Promise.allSettled([promise1, promise2, promise3])
  .then((results) => {
    console.log(results);
    // output: [
    //   { status: 'fulfilled', value: 'Promise 1 resolved' },
    //   { status: 'rejected', reason: Error: Promise 2 rejected },
    //   { status: 'fulfilled', value: 'Promise 3 resolved' }
    // ]
  })
  .catch((err) => {
    console.error(err);
  });

// Promise.race()
Promise.race([promise1, promise2, promise3])
  .then((value) => {
    console.log(value); // output: 'Promise 2 rejected'
  })
  .catch((err) => {
    console.error(err); // output: Error: Promise 2 rejected
  });

// Promise.any()
Promise.any([promise1, promise2, promise3])
  .then((value) => {
    console.log(value); // output: 'Promise 1 resolved'
  })
  .catch((err) => {
    console.error(err); // output: AggregateError: All promises were rejected
  });
```

In this example, we create three promises: `promise1`, `promise2`,

and `promise3`. `promise1` and `promise3` resolve after delays of 1.5 and 1 seconds respectively, while `promise2` rejects after a delay of 500 milliseconds.

We then demonstrate the use of all the Promise methods:

- `Promise.all()` waits for all the promises to resolve or reject, and returns an array of the resolved values of the input promises. In this case, it rejects with the error of `promise2`, since it was the first promise to reject.
- `Promise.allSettled()` waits for all the promises to settle and returns an array of objects representing the status of each promise. In this case, it returns an array of three objects, each containing the status and value or reason of the corresponding promise.
- `Promise.race()` waits for the first promise to settle (either resolve or reject) and returns the value of that promise. In this case, it rejects with the error of `promise2`, since it was the first promise to reject.
- `Promise.any()` waits for the first promise to fulfill (resolve) and returns the value of that promise. In this case, it resolves with the value of `promise1`, since it was the first promise to resolve.

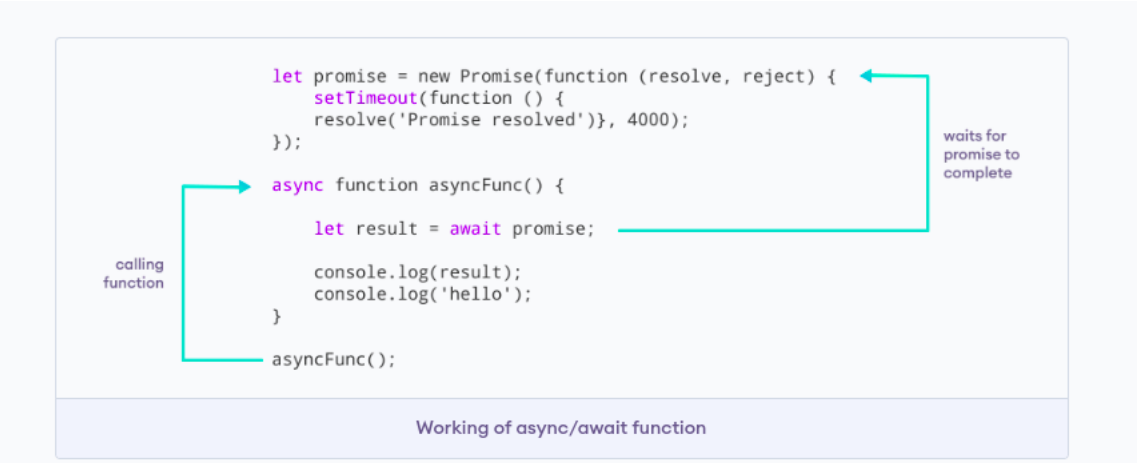
▼ Async/Await

- `async` keyword is keyword that comes with the function to represent that the function is an asynchronous function.

```
async function function_name(param1, ....paramn){
// code
}
```

- `await` keyword is used inside `async` function to wait for the asynchronous operations.
- the syntax is `let result = await promise;`

```
const result = // some time taking operation
async function myFunc(){
let ans = await result;   // waits for the result to arrive
console.log(ans);
}
myFunc();
```



Working of async/await function

`we can only use await inside async function`

- `Benefits`:
  - Code is more readable than using a callback or a promise.
  - Error handling is simpler.
  - Debugging is easier

`Error handling`:

- We use `try/catch` for error handling.

```
const result = // some time taking operation
async function myFunc(){
try{
let ans = await result;   // waits for the result to arrive
console.log(ans);
}
catch(error){
console.error;
}
}
myFunc();
```

If the program runs successfully, it will go to try block, if any error occurs the code inside the catch block is executed.

▼ Map and Set

▼ Map

- It is the new data structure introduced in ES6.
- Similar to objects in javascript and allows us to store elements in key/value pair.

- unlike objects, map can contain objects, functions and other data types as key.

```
// creating a map

const map1 = new Map();    // using the Map() constructor.
console.log(map1);      // returns Map {}
```

- `Insert item to map`:
  - We can use `set()` method to inset element to map.

```
let map1 = new Map();

// insert key-value pair
map1.set('info', {name: 'ashwin', age:22});
console.log(map1);  //  Map(1) {'info', {name: 'ashwin', age:22}}
```

  - we can also use `object` as a key

```
let map1 = new Map();
let obj1 = {}
// insert key-value pair
map1.set(obj1, {name: 'ashwin', age:22});
console.log(map1);  //  Map(1) {{}, {name: 'ashwin', age:22}}
```

- `Accesssing map element`:
  - We can use `get()` method to access the elements.

```
let map1 = new Map();

// insert key-value pair
map1.set('info', {name: 'ashwin', age:22});
console.log(map1.get('info'));
// outpur: {name: 'ashwin', age:22}
```

- `check map elements`: to check if the element present in the map or not.
  - We can use `has()` method to check if the element is in the map or not.

```
let map1 = new Map();

// insert key-value pair
map1.set('info', {name: 'ashwin', age:22});
console.log(map1.has("info");  //  true
```

- `removing element from the map`:
  - we can use `clear()` and `delete()` method to remove elements from the Map.
  - The `delete()` method returns `true` is the specified key/value pair exists and has been removed or else returns `false`.

```
let map1 = new Map();

// insert key-value pair
```

```
map1.set('info', {name: 'ashwin', age:22});
console.log(map1.delete("info"); // returns true
```

- The `clear()` method removes all the key/value pair from the map object.

```
let map1 = new Map();

// insert key-value pair
map1.set('info', {name: 'ashwin', age:22});
map1.clear();    // removes all the items from the map
console.log(map1);
```

- `map size`:

```
map1.size();  // gets the number of elements in a Map
```

- `iterate through a Map`:
  - `for-of` and `forEach()` can be used to iterate through a map

```
let map1 = new Map();
map1.set('name', 'Jack');
map1.set('age', '27');

// looping through Map using for-of
for (let [key, value] of map1) {
    console.log(key + '- ' + value);
}

// looping through Map using forEach()
map1.forEach(function(value, key) {
  console.log(key + '- ' + value)
})
```

- `Iterate over map keys and values`:
  - We can use `keys()` method to get the keys.
  - We can use `values()` method to get the values.

```
let map1 = new Map();
map1.set('name', 'Jack');
map1.set('age', '27');

// looping through the Map to get keys
for (let key of map1.keys()) {
  console.log(key)
}

// looping to get values
for (let values of map1.values()){
console.log(values);
}
```

- `get key/value of map`:
  - `entries()` method can be used.

```
let map1 = new Map();
map1.set('name', 'Jack');
map1.set('age', '27');

// looping through the Map
for (let elem of map1.entries()) {
    console.log(`${elem[0]}: ${elem[1]}`);
}

// output:
// name: Jack
// age: 27
```

`Map vs Object`:

Map	Object
Maps can contain objects and other data types as keys.	Objects can only contain strings and symbols as keys.
Maps can be directly iterated and their value can be accessed.	Objects can be iterated by accessing its keys.
The number of elements of a Map can be determined by `size` property.	The number of elements of an object needs to be determined manually.
Map performs better for programs that require the addition or removal of elements frequently.	Object does not perform well if the program requires the addition or removal of elements frequently.

In JavaScript, objects and functions can be used as keys in another object. When an object or function is used as a key in an object, it is first converted to a string using the `toString()` method. The resulting string is then used as the key in the object.

```
const obj = {
  [{}]: 'value1',
  [function(){}]: 'value2'
};

console.log(obj); // output: { '[object Object]': 'value1', 'function (){}': 'value2' }
```

▼ Weak Map

- It is similar to map. however, it can only contain objects as keys.

```
const weakMap = new WeakMap();
console.log(weakMap); // WeakMap {}

let obj = {};

// adding object (element) to WeakMap
weakMap.set(obj, 'hello');

console.log(weakMap); // WeakMap {{} => "hello"}
```

- If we try to add other data type as key, it throws an error.

```
const weakMap = new WeakMap();

// adding string as a key to WeakMap
weakMap.set('obj', 'hello');    // throws an error
```

`WeakMap Methods:`

- It has `get()`, `set()`, `delete()` and `has()` methods.

```
const weakMap = new WeakMap();
console.log(weakMap); // WeakMap {}

let obj = {};

// adding object (element) to WeakMap
weakMap.set(obj, 'hello');

console.log(weakMap); // WeakMap {{} => "hello"}

// get the element of a WeakMap
console.log(weakMap.get(obj)); // hello

// check if an element is present in WeakMap
console.log(weakMap.has(obj)); // true

// delete the element of WeakMap
console.log(weakMap.delete(obj)); // true

console.log(weakMap); // WeakMap {}
```

- `Unlike Maps, WeakMaps are not iterable`

▼ Sets

- Set is similar to array but it cannot have duplicate value.
- If duplicate values are added, it simply excludes the duplicate value and doesn't throw an error.
- It can have multiple types of value.

```
// creating set
const set1 = new Set(); // empty set
console.log(set1);  // Set {}

// set with multiple set of values
const set2 = new set([1, "hello", {isTrue: true}]);
console.log(set2);
```

- `Accessing Set Elements`:
  - We can access set elements using `values()` method and check if element inside `set` using `has()` method.

```
const set1 = new Set([1,2,3]);

// accessing elements in set
console.log(set1.values());

// checking is the element is present in the set or not
console.log(set1.has(1));  // true
```

- `Adding new element`:
  - we can use `add()` method to add elements in the set.

```
set1.add(4)
set1.add(1)   // excludes as 1 is already present in the set
```

- `Removing element`:
  - we use `clear()` and `delete()` to remove the element from the set.

```
// delete removes specific element from the set
set1.delete(1);    // removes 1 from the set and returns true
// if the element is not present in the set it returns false

// clear removes all the elements from the set
set1.clear();    // removes everything
```

- `Iterating set`:
- `for-of` and `forEach()` methods can be used to iterate through set.

```
const set = new Set([1, 2, 3]);

// looping through Set
for (let i of set) {
    console.log(i);
}
```

▼ Weak Set

- Similar to set.
- It can only contain `objects` unlike set which can have elements of different data types.

```
const weakSet = new WeakSet();
console.log(weakSet); // WeakSet {}

let obj = {
    message: 'Hi',
    sendMessage: true
}

// adding object (element) to WeakSet
weakSet.add(obj);
console.log(weakSet); // WeakSet {{message: "Hi", sendMessage: true}}
```

```
weakSet.add('hello');    // throws an error
```

`Methods`:

- it has `add()`, `delete()`, and `has()` methods.

```
const weakSet = new WeakSet();
console.log(weakSet); // WeakSet {}

const obj = {a:1};

// add to a weakSet
weakSet.add(obj);
console.log(weakSet); // WeakSet {{a: 1}}

// check if an element is in Set
console.log(weakSet.has(obj)); // true
```

```
// delete elements
weakSet.delete(obj);
console.log(weakSet); // WeakSet {}
```

- Similar to WeakMap it `cannot be iterated`

`Set Operations`

- Set doesn't provide with built-in methods for performing union, intersection, difference, etc. operations. However we can perform those operations.

1. `Set Union Operations`:

```
function union(a, b) {
    let unionSet = new Set(a);
    for (let i of b) {
        unionSet.add(i);
    }
    return unionSet
}

// two sets of fruits
let setA = new Set(['apple', 'mango', 'orange']);
let setB = new Set(['grapes', 'apple', 'banana']);

let result = union(setA, setB);

console.log(result);
```

2. `Set intersection Operations`;

```
// perform intersection operation
// elements of set a that are also in set b
function intersection(setA, setB) {
    let intersectionSet = new Set();

    for (let i of setB) {
        if (setA.has(i)) {
            intersectionSet.add(i);
        }
    }
    return intersectionSet;
}

// two sets of fruits
let setA = new Set(['apple', 'mango', 'orange']);
let setB = new Set(['grapes', 'apple', 'banana']);

let result = intersection(setA, setB);

console.log(result);    // Set {"apple"}
```

3. `Set Difference Operations`:

```
function difference(setA, setB) {
    let differenceSet = new Set(setA)
    for (let i of setB) {
        differenceSet.delete(i)
    }
    return differenceSet
}

// two sets of fruits
```

```
let setA = new Set(['apple', 'mango', 'orange']);
let setB = new Set(['grapes', 'apple', 'banana']);

let result = difference(setA, setB);

console.log(result);   // Set {"mango", "orange"}
```

4. `Set Subset Operations`:

```
function subset(setA, setB) {
    for (let i of setB) {
        if (!setA.has(i)) {
            return false
        }
    }
    return true
}

// two sets of fruits
let setA = new Set(['apple', 'mango', 'orange']);
let setB = new Set(['apple', 'orange']);

let result = subset(setA, setB);

console.log(result);        // true
```

▼ for… of Loop

- `for... of` loop allows us to iterate is over iterable objects (arrays, sets, maps, string, etc).

```
// SYNTAX
for (element of iterable){
    // code here
}
```

`element` is the items in the iterable, `iterable` can be array, set, etc.

`for... of with Array`:

```
// array
const students = ['John', 'Sara', 'Jack'];

// using for...of
for ( let element of students ) {

    // display the values
    console.log(element);
}

// output john\n Sara\n Jack
```

`for... of with String`:

```
// string
const string = 'code';

// using for...of loop
for (let i of string) {
    console.log(i);
}

// output: c\n o\n d\n e\n
```

`for... of with sets`:

```
// define Set
const set = new Set([1, 2, 3]);

// looping through Set
for (let i of set) {
    console.log(i);
}
```

`for... of with Map`:

```
// define Map
let map = new Map();

// inserting elements
map.set('name', 'Jack');
map.set('age', '27');

// looping through Map
for (let [key, value] of map) {
    console.log(key + '- ' + value);
}
```

`for... of with generators`:

```
// generator function
function* generatorFunc() {

    yield 10;
    yield 20;
    yield 30;
}

const obj = generatorFunc();

// iteration through generator
for (let value of obj) {
    console.log(value);
}
```

`for...of vs for...in`:

for...of	for...in
The `for...of` loop is used to iterate through the values of an iterable.	The `for...in` loop is used to iterate through the keys of an object.
The `for...of` loop cannot be used to iterate over an object.	You can use `for...in` to iterate over an iterable such arrays and strings but you should avoid using `for...in` for iterables.

▼ Closure

-

▼ This

- it refers to the object where it is called.

`This inside Global Scope` :

- When `this` is used alone. it refers to the global object. (`window` object in browsers).

```
let a = this;
console.log(a);  // Window {}

this.name = 'Sarah';
console.log(window.name); // Sarah
```

here `this.name` is same as `window.name`.

`This inside a function`:

- When `this` is used inside a function, it refers to the global object `window`.

```
function greet() {

    // this inside function
    // this refers to the global object
    console.log(this);
}

greet(); // Window {}
```

`This inside constructor function`:

- `this` refers to the object inside which it is used.

```
function Person(){
this.name = 'jack';
console.log(this);   // Person {name: 'jack'}
}

let person1 = new Person();
console.log(person1.name);  // jack
```

here, `this` refers to the `person1` object. so `person1.name` gives `jack`.

`this inside object method`:

- when used inside of object, it refers to the object it lies within.

```
const person = {
    name : 'Jack',
    age: 25,
    // this inside method
    // this refers to the object itself
    greet() {
        console.log(this);
        console.log(this.name);
    }
}

person.greet();
```

`this inside inner function`:

- When used inside inner function, `this` refers to global object i.e `window`.

`this Inside Arrow Function`:

- inside arrow function, `this` refers to parent scope.

```
const greet = {
    name: 'Jack',

    // method
    sayHi () {
```

```
        let hi = () => console.log(this.name);   // refers to the greet scope
        hi();
    }
}

greet.sayHi(); // Jack
```

`this inside function with strict mode`:

- When `this` is used in a function with strict mode, `this` is `undefined`.

```
'use strict';
this.name = 'Jack';
function greet() {

    // this refers to undefined
    console.log(this);
}
greet(); // undefined
```

- BUT, we can use `function.call()` method to use `this` inside a function with strict mode.

```
'use strict';
this.name = 'Jack';

function greet() {
    console.log(this.name);
}

greet.call(this); // Jack
```

▼ IsNaN and Intl

`isNaN()`:

- It is a function that determines whether a value is NaN (Not a Number). The `isNaN()` function takes a single argument and returns `true` if the argument is NaN, and `false` otherwise. If the argument is not a number, the `isNaN()` function first tries to convert it to a number using the `Number()` function before checking if it is NaN.

```
console.log(isNaN(123)); // output: false
console.log(isNaN('hello')); // output: true
console.log(isNaN('123')); // output: false
console.log(isNaN(true)); // output: false
console.log(isNaN(undefined)); // output: true
console.log(isNaN(null)); // output: false
```

In this example, we use the `isNaN()` function to check whether various values are NaN. For example, `isNaN('hello')` returns `true` because the string `'hello'` cannot be converted to a number, while `isNaN('123')` returns `false` because the string `'123'` can be
converted to the number `123`

`Intl`:

- It is an object that provides internationalization features in JavaScript. The `Intl` object contains several properties and methods for formatting numbers, dates, and times in a locale-specific way.

```
const date = new Date();
const number = 12345.6789;

console.log(new Intl.DateTimeFormat('en-US').format(date)); // output: 4/25/2023
console.log(new Intl.NumberFormat('en-US').format(number)); // output: 12,345.679
```

In this example, we use the `Intl.DateTimeFormat()` method to format the current date in the `'en-US'` locale, and the `Intl.NumberFormat()` method to format the number `12345.6789` in the `'en-US'` locale. The resulting output is formatted according to the conventions of the `'en-US'` locale, such as using the month-day-year date format, and using a comma as the thousands separator in the formatted number.

Overall, `isNaN()` and `Intl` are powerful built-in features of JavaScript that provide useful functionality for working with numbers and internationalization.

▼ Hoisting

- It is the behavior of javascript where a variable can be used before declaration.

```
console.log(test);
var test;
```

- The above program works and the output will be `undefined`. as test has no value.

1. `hoisting in var`:

   ```
   console.log(a);
   var a = 5;         // undefined
   ```

   The above program runs as;

   ```
   var a;
   console.log(a);    // so the value of a is undefined.
   a = 5;
   ```

2. `hoisting in let`;

   - If a variable is hoisted with `let` keyword. the variable is not hoisted.

   ```
   a = 5;
   console.log(a);
   let a; // error
   ```

   - While using `let`, the variable must be declared first.

3. `hoisting in function`:

   - A function can be called before declaring it.

   ```
   greet();
   function greet(){
   console.log("hi, there.");       // hi, there.
   }
   ```

   However, if the function is used as an expression, an error occurs because only the declarations are hoisted. defining function using let or var has same hoisting behavior as of let and var.

▼ OOP

▼ 4 Pillars of OOP.

1. Inheritance
2. Encapsulation
3. Polymorphism
4. Abstraction

▼ Object

- Object is the unique entity that contains properties and methods. For example: car is the real life object, which has some characteristics like color, type, model, etc. The characteristics of objects are called properties and actions are called methods.
- It is a instance of a class.
- Object can be created in two ways in javaScript:
  1. Object Literal
  2. Object Constructor

    - using `object literal` :

```
let person = {
  first_name:'abc',
  last_name: 'def',

  //method
  getFunction : function(){
    return (`The name of the person is
    ${person.first_name} ${person.last_name}`)
  },
  //object within object
  phone_number : {
    mobile:'12345',
    landline:'6789'
  }
}
console.log(person.getFunction());
console.log(person.phone_number.landline);
```

    - using `object constructor`:

```
function person(first_name,last_name){
       this.first_name = first_name;
       this.last_name = last_name;
   }
   // Creating new instances of person object
   let person1 = new person('abc','def');
   let person2 = new person('def','abc');

   console.log(person1.first_name);
   console.log(`${person2.first_name} ${person2.last_name}`);
```

▼ Class

- Class are the blueprint of an object. Class can have many objects because class is a template while objects are instances of class.

```
//defining the class in ES6
class Vehicle{
  constructor(name, maker, engine){
      this.name = name;
      this.maker = maker;
      this.engine = engine;
  }

  getDetails(){
      retrun (`The name of the vehicle is ${this.name}.`);
  }
}

// creating a object
let bike1 = new Vehicle('abc','def','ghi');

console.log(bike1.name);
console.log(bike1.getDetails());
```

```
// defining class traditionally
function Vehicle(name,maker,engine){
        this.name = name,
        this.maker = maker,
        this.engine = engine
    };

// prototype garyo bhane yo function sabai instance lai same hunxa ani available hunxa
    Vehicle.prototype.getDetails = function(){
        console.log('The name of the bike is '+ this.name);
    }

    let bike1 = new Vehicle('Hayabusa','Suzuki','1340cc');
    let bike2 = new Vehicle('Ninja','Kawasaki','998cc');

    console.log(bike1.name);
    console.log(bike2.maker);
    console.log(bike1.getDetails());
```

When a method is added to the constructor function, a new copy of the method is created for each instance of the object. This can be inefficient if you have many instances of the object, because it can take up more memory than necessary.

On the other hand, when a method is added to the prototype of the constructor function, the method is shared by all instances of the object. This means that only one copy of the method is created in memory, regardless of how many instances of the object are created. This can be much more memory efficient, particularly if you have many instances of the object.

▼ Abstraction

- Abstraction means displaying only essential information and hiding the details.
- Data abstraction refers to providing only essential information about the data to the outside world. hiding the background details or implementation.

```
class BankAccount {
  constructor(balance) {
    this.balance = balance;
  }

  deposit(amount) {
    this.balance += amount;
  }

  withdraw(amount) {
    if (amount > this.balance) {
      console.log("Insufficient funds");
    } else {
      this.balance -= amount;
    }
  }
```

```
  getBalance() {
    return this.balance;
  }
}
```

- here, all the implementation is hidden to the user, user just needs to know how to call the methods to interact with the bank account.
- Example:

```
// user can open bank account like:
const myAcc = new BankAccount(1000);

// user can deposit money:
myAcc.deposit(500);

// can check balance:
console.log(myAcc.getBalance());
```

▼ Encapsulation

- The process of wrapping properties and functions within a single unit is encapsulation.
- involves hiding the details of an object's implementation from outside code, and only exposing a public interface for interacting with the object. This can help to protect the object's data from being modified accidentally or maliciously.

```
class BankAccount {

  #privVar;   // we can define private properties in class

  constructor(balance) {
    let _balance = balance; // private property
    this.#privVar = 2;    // private property

    this.deposit = function(amount) { // public method
      _balance += amount;
    };

    this.withdraw = function(amount) { // public method
      if (amount > _balance) {
        console.log("Insufficient funds");
      } else {
        _balance -= amount;
      }
    };

    this.getBalance = function() { // public method
      return _balance;
    };
  }

  #privateMethod(){             // we can create a private method like this
  // private method
  }
}
```

- `there is no access modifiers in javascript` but we can restrict the scope of variable within the class.
- here in above example, `_balance` is defined inside the constructor and is not accessible outside the constructor function. And cannot be accessed directly from outside the class.
- let, const, and closure can be used to achieve this behavior.

▼ Inheritance

- It allows objects to inherit properties and methods from a parent object, and to extend or modify as needed.
- we use `entend` keyword.

```
class Animal{
  constructor(name){
    this.name = name;
  }

  speak(){
    console.log(`this is speak function`);
  }
}

// child clas
class Dog extends Animal{
  constructor(name){
    super(name);   // calling the parent class constructor method and properties.
  }

  speak(){          // overriding the speak() method
    console.log(`this is dog`);
  }
}

// Create objects
const a = new Animal('Animal');
const d = new Dog('Dog');

// Call methods
a.speak(); // Output: "This is speak function"
d.speak(); // Output: "this is dog"
```

▼ Polymorphism

- It means the same function with different signatures.

`Overriding`:

- same method or instance with same parameters and all but has different implementation.

```
constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {
  constructor(name) {
    super(name);
  }

  speak() {              // this is the overrided method
    console.log(`${this.name} barks.`);
  }
}
```

`Oevrloading`:

- Overloading is not directly supported in JavaScript because of its dynamic and loosely-typed nature. Unlike strongly-typed languages like Java or C++, JavaScript does not require explicit type declarations for variables, and allows variables to change their type at runtime.
- latest defined methods matra chalxa mathi ko sab ignore garxa.

▼ Design Patterns

- Design patterns are typical solutions to common problems in software design.

- Each pattern is like a blueprint that we can customize to solve a particular design problem in our code.
- Pattern is `not a code` but a general concept for solving particular problem.
- We can follow pattern details and implement a solution that suits the realities of our program and are reusable.

Design patterns are a set of general solutions to common software design problems. They provide a proven way to solve problems that can arise during software development, and can be used to improve the quality and maintainability of code. There are three main types of design patterns:

- Patterns can be categorized by their intent, or purpose.

1. `Creational patterns:` These patterns are used to create objects in a system, and can help to ensure that the objects are created in a consistent and reusable way. Examples include Singleton and Factory patterns.
2. `Structural patterns:` These patterns are used to define the structure of objects and classes in a system, and can help to ensure that the system is organized in a maintainable way. Examples include Adapter and Bridge patterns.
3. `Behavioral patterns:` These patterns are used to define the behavior of objects and classes in a system, and can help to ensure that the system is flexible and extensible. Examples include Observer and State patterns.

By using design patterns, developers can save time and improve the quality of their software. However, it's important to remember that design patterns are not a silver bullet, and should be used judiciously.

- `What does the pattern consists of`:
  - `Intent`: defines both problem and solutions briefly.
  - `Motivation`: further explains problem and solution and reason for using the pattern.
  - `Structure`: shows each part of pattern and how they are related.
  - `code example`: an example implementation of the pattern in code.

- `History`:

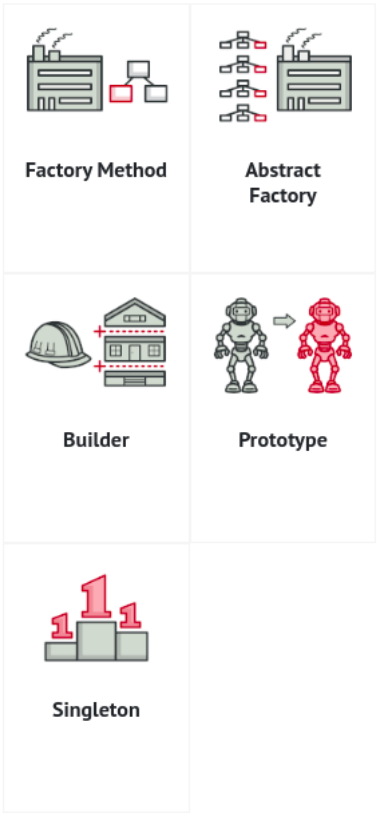
  The concept of patterns in design was first introduced by Christopher Alexander, who described a "language" for designing the urban environment in his book "A Pattern Language". The idea was later applied to programming by the Gang of Four, who published the book "Design Patterns: Elements of Reusable Object-Oriented Software" featuring 23 patterns for solving problems in object-oriented design. Since then, many other patterns have been discovered and applied in various programming fields.

▼ Creational Patterns

- It provides various object creation mechanism that increases flexibility and reuse of existing code.
- It is a way of creating objects in a flexible and reusable way.
- It provides a set of guidelines for creating objects that are independent of the specifics of their implementation, allowing developers to create object that are easily maintained, tested and extended.

Types

1. Factory Method
2. Abstract Factory
3. Builder
4. Prototype
5. Singleton



▼ Factory Method

- It is the way of creating objects in a superclass, but allowing subclasss to determine type of object that will be created.
- It is useful when we need to create objects that have similar behaviors but different implementation.

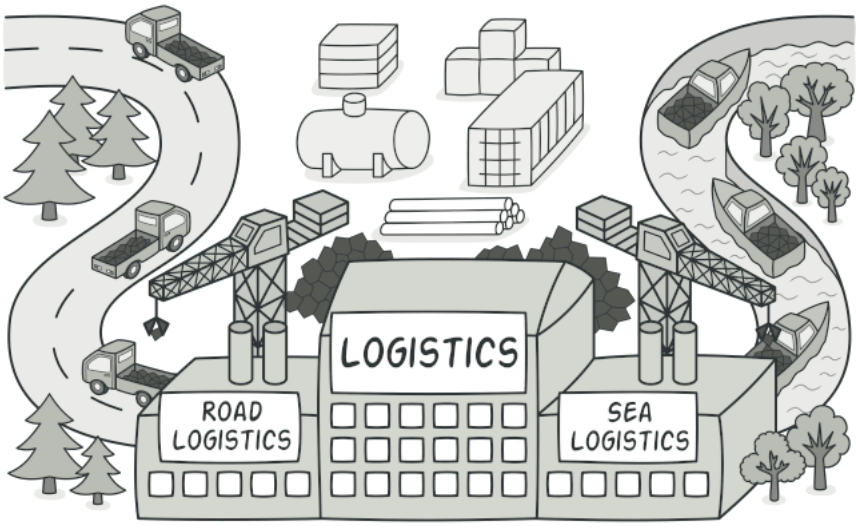
`Key Elements:`

1. `Product`: It is the interface or abstract class that defines behavior of the object created.
2. `Concrete product`: It is the specific implementation of product interface, each with its unique behavior.
3. `creator`: It is the abstract class that defines factory method, which is responsible for creating objects. It may also contain other methods that use products created by factory method.

4. `concrete creator:` it is the subclass of creator class, overrides the base methods so it returns different type of product.

▼ Intent

- Intent is to provide way of creating objects in flexible and extensible way, by allowing subclass to determine the type of object that will be created. This allows for the creation of objects that have similar behaviors but different implementation, and make it easy to add new types of object without modifying existing code.



▼ Problem

For a game app, let's say we have multiple characters. each with own unique abilities, if the creation of the each character is tightly coupled to the implementation details in the game. adding a new character would require modifying existing code, which is time-consuming and error-prone.

```
// PROBLEM
public class Character {
    private String type;

    public Character(String type) {
        this.type = type;
    }

    public void attack() {
        if (type.equals("Warrior")) {
            // Warrior's attack behavior
        } else if (type.equals("Mage")) {
            // Mage's attack behavior
        }
    }

    public void defend() {
        if (type.equals("Warrior")) {
            // Warrior's defend behavior
        } else if (type.equals("Mage")) {
            // Mage's defend behavior
        }
    }
}
```

Here, in the above example, adding a new character requires modifying existing character class, which violates the `Open/Close` principle. *"classes should be open for extensions but closed for modification".*

We need to modify the attack and defend methods to include new conditional statements for new character, and add new instance variable or methods to character class. Which results in more complexity, no flexibility and extensibility, more conditional statements and nasty code.

▼ Solution

```
// Product interface - defines the behavior of all characters in the game
public interface Character {
    void attack();
    void defend();
}

// Concrete Products - specific character classes, each with its own unique abilities and behaviors
public class Warrior implements Character {
    @Override
    public void attack() {
        // Warrior's attack behavior
    }

    @Override
    public void defend() {
        // Warrior's defend behavior
    }
}

public class Mage implements Character {
    @Override
    public void attack() {
        // Mage's attack behavior
    }

    @Override
    public void defend() {
        // Mage's defend behavior
    }
}

// Creator - abstract class that defines the Factory Method
public abstract class CharacterCreator {
    public abstract Character createCharacter();
}

// Concrete Creators - subclasses of the Creator class, each of which implements the Factory Method to create a specific type
public class WarriorCreator extends CharacterCreator {
    @Override
    public Character createCharacter() {
        return new Warrior();
    }
}

public class MageCreator extends CharacterCreator {
    @Override
    public Character createCharacter() {
        return new Mage();
    }
}
```

here, the character interface and warrior and mage classes are same as in the previous example.

The difference is that here we have `creator` and `Concrete creator`

- Instead of creating a new character directly we can use `createCharacter()` method to create a new character.
- This allows for the easy addition of new types of character without modifying the existing code.
- To add new character here we can just add:

```
public class Rogue implements Character {
    @Override
    public void attack() {
        // Rogue's attack behavior
    }

    @Override
    public void defend() {
        // Rogue's defend behavior
    }
}

public class RogueCreator extends CharacterCreator {
```

```
    @Override
    public Character createCharacter() {
        return new Rogue();
    }
}
CharacterCreator rogueCreator = new RogueCreator();
Character rogue = rogueCreator.createCharacter();
```

It allows for easy addition of new types of objects without modifying existing code. And is more flexible and extensible.

▼ Application

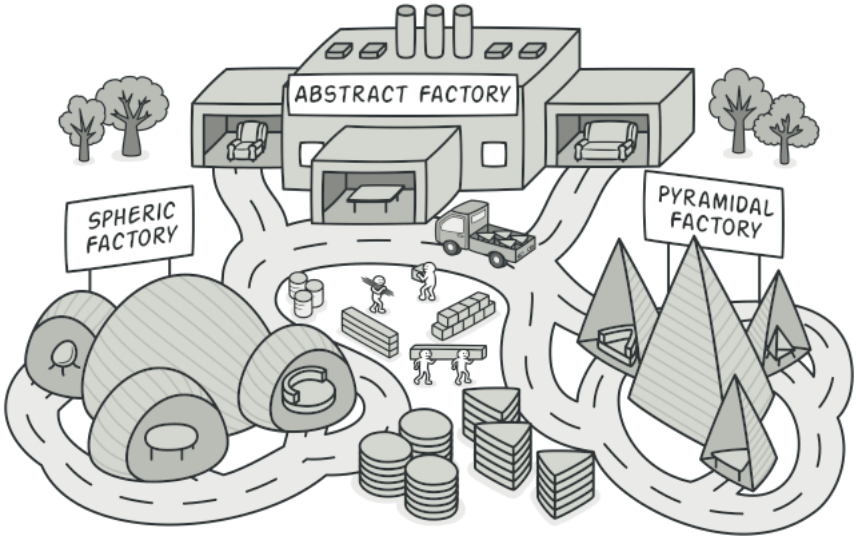
1. Use the Factory Method when you don't know beforehand the exact types and dependencies of the objects your code should work with.
2. Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components.
3. Use the Factory Method when you want to save system resources by reusing existing objects instead of rebuilding them each time.

▼ Abstract Factory

- To create families of related object without specifying their concrete classes.

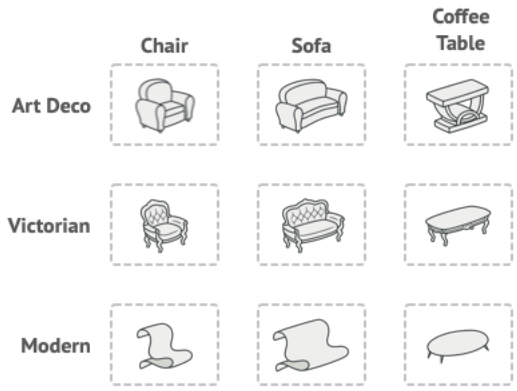
▼ Intent

- To provide an interface for creating families of related or dependent objects without specifying their concrete class. OR,
- It allows us to create family of related objects without having to worry about specific implementation details of each object.
- flexible, modular and easy to extend.



▼ Problem

- Imagine we're creating a furniture shop simulator, where code consists of classes that represents:
    1. family of related products: `chair` + `sofa` + `table`
    2. special variant of this family: `modern` , `vicrorian` , `artDeco`



- We need a way to create individual furniture object so that they match other of same family.
- Also, we don't want to change existing code when adding new products or families of products to the program.

```
// Product interface
interface Furniture {
    void sit();
}

// Concrete product classes
class ModernChair implements Furniture {
    @Override
    public void sit() {
        System.out.println("Sitting on a modern chair");
    }
}

class ModernSofa implements Furniture {
    @Override
    public void sit() {
        System.out.println("Sitting on a modern sofa");
    }
}

class ModernTable implements Furniture {
    @Override
    public void sit() {
        System.out.println("Sitting at a modern table");
    }
}

class VictorianChair implements Furniture {
    @Override
    public void sit() {
        System.out.println("Sitting on a Victorian chair");
    }
}

class VictorianSofa implements Furniture {
    @Override
    public void sit() {
        System.out.println("Sitting on a Victorian sofa");
    }
}

class VictorianTable implements Furniture {
    @Override
    public void sit() {
        System.out.println("Sitting at a Victorian table");
    }
}

class ArtDecoChair implements Furniture {
    @Override
    public void sit() {
        System.out.println("Sitting on an Art Deco chair");
```

```
    }
}

class ArtDecoSofa implements Furniture {
    @Override
    public void sit() {
        System.out.println("Sitting on an Art Deco sofa");
    }
}

class ArtDecoTable implements Furniture {
    @Override
    public void sit() {
        System.out.println("Sitting at an Art Deco table");
    }
}
```

- This code is harder to understand and maintain, add or change existing products. New method should be added everytime.
- violates DRY concept for `sit()` method.

▼ Solution

```
// Product interfaces
interface Chair {
    void sit();
}

interface Sofa {
    void sit();
}

interface Table {
    void sit();
}

// Concrete product classes
class ModernChair implements Chair {
    @Override
    public void sit() {
        System.out.println("Sitting on a modern chair");
    }
}

class ModernSofa implements Sofa {
    @Override
    public void sit() {
        System.out.println("Sitting on a modern sofa");
    }
}

class ModernTable implements Table {
    @Override
    public void sit() {
        System.out.println("Sitting at a modern table");
    }
}

class VictorianChair implements Chair {
    @Override
    public void sit() {
        System.out.println("Sitting on a Victorian chair");
    }
}

class VictorianSofa implements Sofa {
    @Override
    public void sit() {
        System.out.println("Sitting on a Victorian sofa");
    }
}

class VictorianTable implements Table {
    @Override
    public void sit() {
        System.out.println("Sitting at a Victorian table");
```

```
    }
}

// Abstract factory interface
interface FurnitureFactory {
    Chair createChair();
    Sofa createSofa();
    Table createTable();
}

// Concrete factory classes
class ModernFurnitureFactory implements FurnitureFactory {
    @Override
    public Chair createChair() {
        return new ModernChair();
    }

    @Override
    public Sofa createSofa() {
        return new ModernSofa();
    }

    @Override
    public Table createTable() {
        return new ModernTable();
    }
}

class VictorianFurnitureFactory implements FurnitureFactory {
    @Override
    public Chair createChair() {
        return new VictorianChair();
    }

    @Override
    public Sofa createSofa() {
        return new VictorianSofa();
    }

    @Override
    public Table createTable() {
        return new VictorianTable();
    }
}

// Client code
public class FurnitureShop {
    public void sitOn(Chair chair) {
        chair.sit();
    }

    public void sitOn(Sofa sofa) {
        sofa.sit();
    }

    public void sitOn(Table table) {
        table.sit();
    }

}
```

- if we want to add new item here, we can add as:

```
interface Bookshelf {
    void storeBooks();
}

class ModernBookshelf implements Bookshelf {
    @Override
    public void storeBooks() {
        System.out.println("Storing books in a modern bookshelf");
    }
}

// add method to furniture factory
interface FurnitureFactory {
    Chair createChair();
    Sofa createSofa();
```

```
    Table createTable();
    Bookshelf createBookshelf();
}

//implement new method in concrete factory
class ModernFurnitureFactory implements FurnitureFactory {
    // ...

    @Override
    public Bookshelf createBookshelf() {
        return new ModernBookshelf();
    }
}

class VictorianFurnitureFactory implements FurnitureFactory {
    // ...

    @Override
    public Bookshelf createBookshelf() {
        return new VictorianBookshelf();
    }
}
```

▼ Structure

1. `Abstract Products:` declare interface for set of distinct but related products which make up a product family.
2. `Concrete Products:` various implementation of abstract products, grouped by variants, all abstract products must me implemented in all variants.
3. `Abstract Factory:` declares a set of methods for creating each of abstract products.
4. `Concrete Factories`: implement creation of abstract factory. Each concrete factory corresponds to specific variant of of products and creates only those product variant.

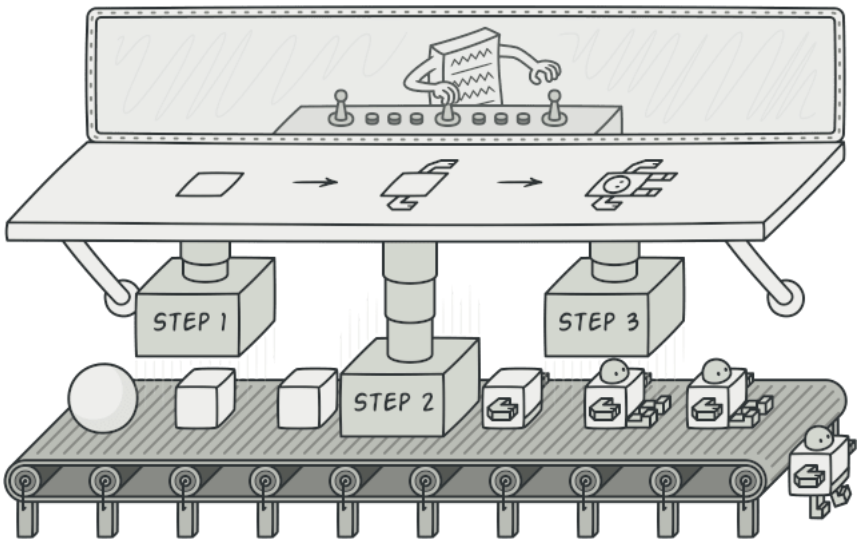
▼ Application

1. When you need to create families of related objects that are designed to work together.
2. When you want to provide a high level of abstraction for creating objects.
3. When you want to provide multiple implementations of a set of related objects.

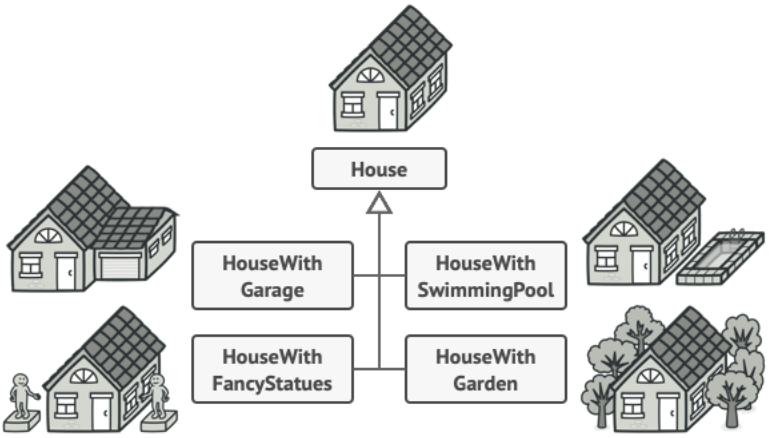
▼ Builder

▼ Intent

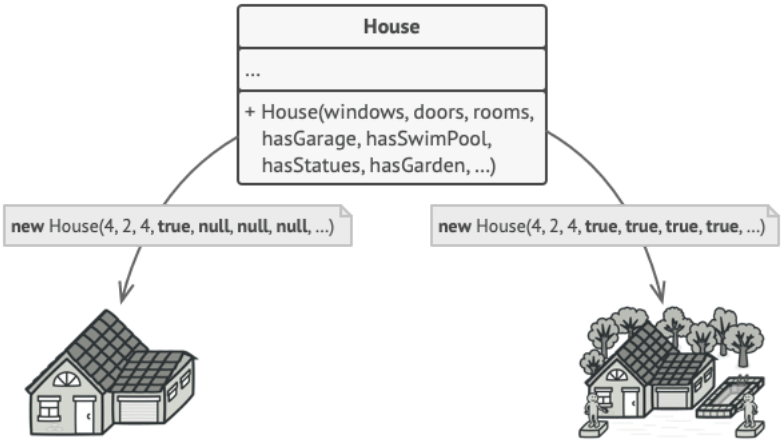
- It helps to create complex objects step by step.
- the pattern allows us to produce different type and representation of an object using same construction code.



▼ Problem



- suppose we want to create a house object
  if we want a bigger house with a backyard and other goodies like (a heating system, plumbing, and electrical wiring)
- the simplest solution is the extends the base house (end up creating a large number of subclasses)
- the following answer is to create a giant constructor right in the base House class with all the possible parameters.



- most cases most of the parameters will be unused, making **the constructor calls pretty ugly**. For instance, only a fraction of houses have swimming pools, so the parameters related to swimming pools will be useless nine times out of ten.

```
public class House {
    private int size;
    private boolean hasBackyard;
    private boolean hasHeating;
    private boolean hasPlumbing;
    private boolean hasElectricalWiring;

    public House(int size) {
        this.size = size;
    }

    public int getSize() {
        return size;
    }

    public boolean hasBackyard() {
        return hasBackyard;
    }

    public boolean hasHeating() {
        return hasHeating;
    }

    public boolean hasPlumbing() {
        return hasPlumbing;
    }

    public boolean hasElectricalWiring() {
        return hasElectricalWiring;
    }
}

public class HouseWithBackyard extends House {
    public HouseWithBackyard(int size) {
        super(size);
        this.hasBackyard = true;
    }
}

public class HouseWithHeating extends House {
    public HouseWithHeating(int size) {
        super(size);
        this.hasHeating = true;
    }
}

public class HouseWithPlumbing extends House {
    public HouseWithPlumbing(int size) {
        super(size);
        this.hasPlumbing = true;
    }
```

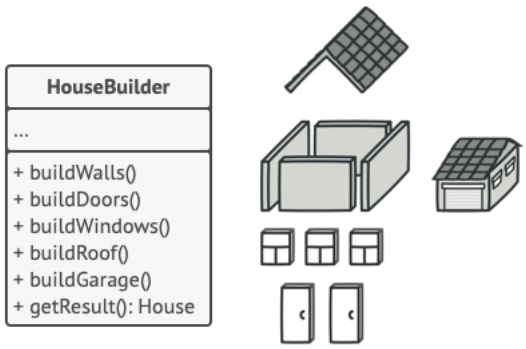
```
}

public class HouseWithElectricalWiring extends House {
    public HouseWithElectricalWiring(int size) {
        super(size);
        this.hasElectricalWiring = true;
    }
}

public class HouseWithBackyardHeatingPlumbingAndElectricalWiring extends House {
    public HouseWithBackyardHeatingPlumbingAndElectricalWiring(int size) {
        super(size);
        this.hasBackyard = true;
        this.hasHeating = true;
        this.hasPlumbing = true;
        this.hasElectricalWiring = true;
    }
}
```

Here, for every requirement we have to create many `subclasses.` so we use builder class.

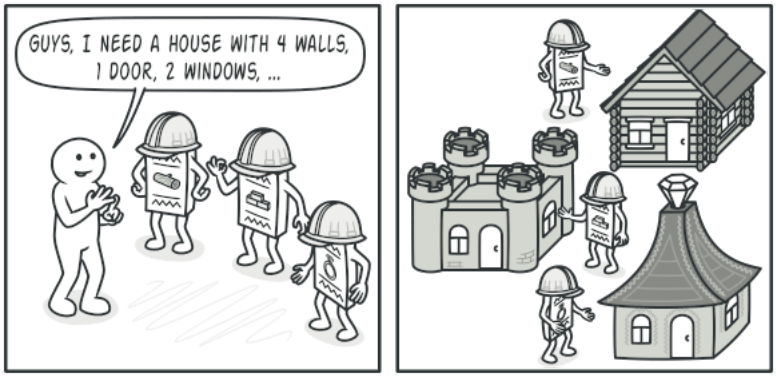
Also constructor may look ugly.

▼ Solution

- extract the object construction code out of its own class and move it to separate objects called builders.



- pattern organizes object construction into a set of steps (`buildWalls`, `buildDoor`, etc.). To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps.
  - Some of the construction steps require different implementations when we need to build various representations of the product.
  In this case, you can create several different builder classes that implement the same set of building steps, but in a different manner.

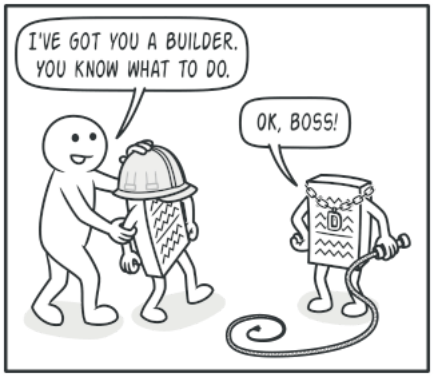


```
public class House {
    private int size;
    private boolean hasBackyard;
    private boolean hasHeating;
    private boolean hasPlumbing;
    private boolean hasElectricalWiring;

    private House(HouseBuilder builder) {
        this.size = builder.size;
        this.hasBackyard = builder.hasBackyard;
        this.hasHeating = builder.hasHeating;
        this.hasPlumbing = builder.hasPlumbing;
        this.hasElectricalWiring = builder.hasElectricalWiring;
    }

    public int getSize() {
        return size;
    }

    public boolean hasBackyard() {
        return hasBackyard;
    }

    public boolean hasHeating() {
        return hasHeating;
    }

    public boolean hasPlumbing() {
        return hasPlumbing;
    }

    public boolean hasElectricalWiring() {
        return hasElectricalWiring;
    }

    public static class HouseBuilder {
        private int size;
        private boolean hasBackyard;
        private boolean hasHeating;
        private boolean hasPlumbing;
        private boolean hasElectricalWiring;

        public HouseBuilder(int size) {
            this.size = size;
        }

        public HouseBuilder hasBackyard(boolean hasBackyard) {
            this.hasBackyard = hasBackyard;
            return this;
        }

        public HouseBuilder hasHeating(boolean hasHeating) {
            this.hasHeating = hasHeating;
            return this;
        }

        public HouseBuilder hasPlumbing(boolean hasPlumbing) {
            this.hasPlumbing = hasPlumbing;
            return this;
        }

        public HouseBuilder hasElectricalWiring(boolean hasElectricalWiring) {
            this.hasElectricalWiring = hasElectricalWiring;
            return this;
        }

        public House build() {
            return new House(this);
        }
    }
}

public class HouseDirector {
    public House buildStandardHouse() {
        return new House.HouseBuilder(1500)
                .hasHeating(true)
                .hasPlumbing(true)
                .hasElectricalWiring(true)
                .build();
```

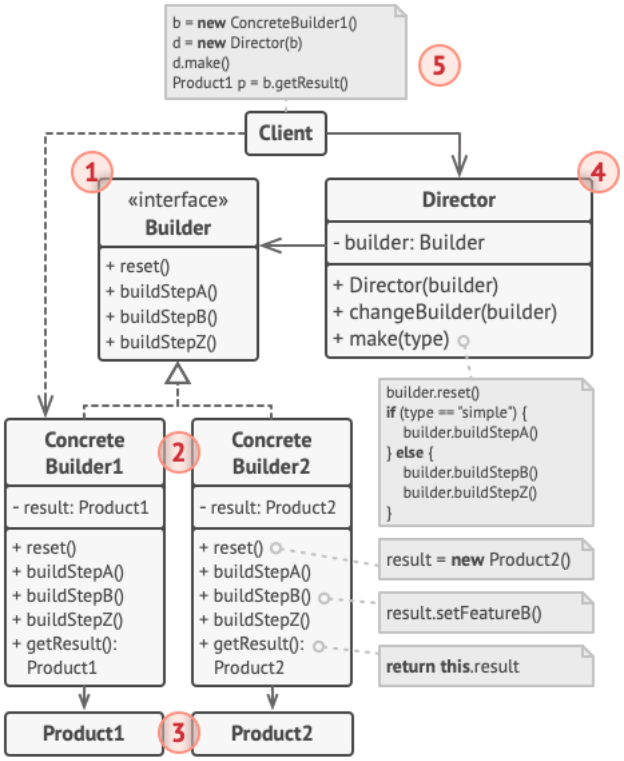
```
    }

    public House buildHouseWithBackyard() {
        return new House.HouseBuilder(1800)
                .hasBackyard(true)
                .hasHeating(true)
                .hasPlumbing(true)
                .hasElectricalWiring(true)
                .build();
    }

    public House buildLuxuryHouse() {
        return new House.HouseBuilder(2500)
                .hasBackyard(true)
                .hasHeating(true)
                .hasPlumbing(true)
                .hasElectricalWiring(true)
                .build();
    }
}
```

▼ Director

- we can extract a series of calls to the builder steps that we use to construct a product into a separate class called director.
- It defines order in which to execute the building steps while builder provides implementation for those steps.
- director class is not strictly necessary, we can directly call steps in specific order from the client code. But we can reuse director class.



▼ Structure



1. The **Builder** interface declares product construction steps that are common to all types of builders.
2. **Concrete Builders** provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.
3. **Products** are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.
4. The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.
5. The **Client** must associate one of the builder objects with the director. Usually, it's done just once, via parameters of the director's constructor. Then the director uses that builder object for all further construction. However, there's an alternative approach for when the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.

▼ Application

1. Use the Builder pattern to get rid of a "telescoping constructor".
2. Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses).
3. The Builder pattern lets you construct products step-by-step. You could defer execution of some steps without breaking the final product. You can even call steps recursively, which comes in handy when you need to build an object tree.

▼ Prototype

▼ Intent

- If we want to copy something and make some changes we want.
- a way of creating new objects based on existing ones.

- This can be useful when creating new objects is time-consuming or difficult, and when the new objects have similar characteristics to existing ones. By copying or cloning existing objects, we can save time and resources, and also avoid errors that may occur during the creation process.

▼ Problem

- if we want to copy of an object, first we have to create new object of same class and go through all fields of original object and copy their values over to new object.
- But there's a catch. Not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself.
- Since we need to know the object's class to create its duplicate, our code becomes dependent on that class.

```
// Problem

class Car {
  constructor(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
  }

  start() {
    console.log('Starting the ' + this.make + ' ' + this.model);
  }
}

// Create new car objects
const car1 = new Car('Toyota', 'Camry', '2021');
const car2 = new Car('Honda', 'Accord', '2022');

// Test the objects
car1.start(); // Starting the Toyota Camry
car2.start(); // Starting the Honda Accord
```

By creating car objects in this way, we are not using the Prototype pattern to create new objects based on a common prototype object. Instead, we are creating new objects by instantiating a class using a constructor function. This can lead to code duplication
if we need to create new classes for different types of objects, and can also make the code less flexible and extensible.

In the class syntax, the `constructor` function is used to define the properties of the class, and any methods (such as `start()`) are defined within the class body. When a new object is instantiated from the class using the `new` keyword, it inherits the properties and methods defined in the class.

▼ Solution

```
// Solution

// Define a prototype object
const carPrototype = {
  make: '',
  model: '',
  year: '2023',
  start: function() {
    console.log('Starting the ' + this.make + ' ' + this.model);
  }
};

// Create a new object from the prototype
const car1 = Object.create(carPrototype);
car1.make = 'Toyota';
car1.model = 'Camry';
car1.year = '2021';

// Create another object from the prototype
const car2 = Object.create(carPrototype);
```

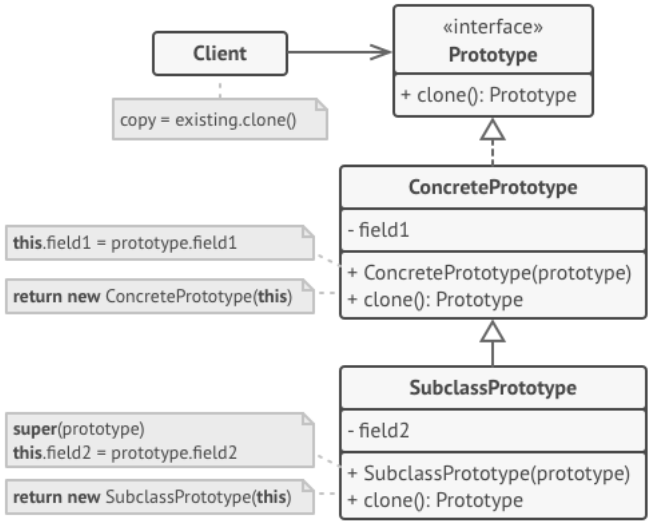
```
// Test the objects
car1.start(); // Starting the Toyota Camry
car2.start(); // Starting the
```

the Prototype pattern allows us to share common properties and methods across multiple instances of an object, which can make the code more efficient and easier to maintain.

Additionally, since each instance of the object shares the same prototype object, any changes made to the prototype object will be reflected in all of the instances. This can make the code more flexible and extensible, and allow for more advanced features such as inheritance and polymorphism.

mathi ko example ma class wala ma, car1 ko start change garyo bhane tesko matra change hunthyo tei ho farak.

▼ Structure



▼ Singleton

▼ Intent

- It is the design pattern that ensures that a class has only one instance of same kind while providing global access point to the instance.

▼ Problem

- If a program has multiple instance of similar kind, it may cause issues like:
  - Resource Wastage
  - Inconsistent Behavior

```
class DatabaseConnection {
   constructor(connectionString) {
       this.connectionString = connectionString;
       this.connection = null;
   }

   getConnection() {
       if (!this.connection) {
           this.connection = DriverManager.getConnection(this.connectionString);
```

```
        }
        return this.connection;
    }
}

const dbConnection1 = new DatabaseConnection('jdbc:mysql://localhost:3306/mydb');
const dbConnection2 = new DatabaseConnection('jdbc:mysql://localhost:3306/otherdb');
```

- the class has a public constructor that takes a connection string as a parameter.
- The `getConnection()` method returns the connection object, creating it if it doesn't exist yet. This implementation allows clients to create multiple instances of the `DatabaseConnection` class, each with its own connection object.

▼ Solution

All implementations of the Singleton have these two steps in common:

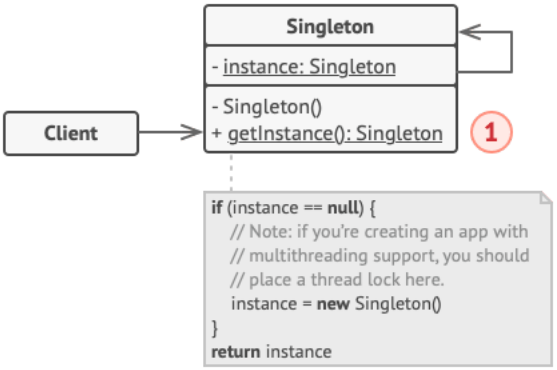
- Make the default constructor private, to prevent other objects from using the `new` operator with the Singleton class.
- Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

If your code has access to the Singleton class, then it’s able to call the Singleton’s static method. So whenever that method is called, the same object is always returned.

```
class DatabaseConnection {
    constructor(connectionString) {
        this.connectionString = connectionString;
        this.connection = null;
    }

    static getInstance(connectionString) {
        if (!DatabaseConnection.instance) {
            DatabaseConnection.instance = new DatabaseConnection(connectionString);
        }
        return DatabaseConnection.instance;
    }

    getConnection() {
        if (!this.connection) {
            this.connection = DriverManager.getConnection(this.connectionString);
        }
        return this.connection;
    }
}

const dbConnection1 = DatabaseConnection.getInstance('jdbc:mysql://localhost:3306/mydb');
const dbConnection2 = DatabaseConnection.getInstance('jdbc:mysql://localhost:3306/otherdb');
```

- we define a static `getInstance()` method that returns the same instance of the `DatabaseConnection` class every time it is called, ensuring that only one connection object is created and used throughout the program.
- The `getInstance()` method checks whether an instance of the `DatabaseConnection` class already exists, and creates a new one if it doesn't.
- The `getConnection()` method returns the connection object, creating it if it doesn't exist yet.

▼ Structure



1. The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

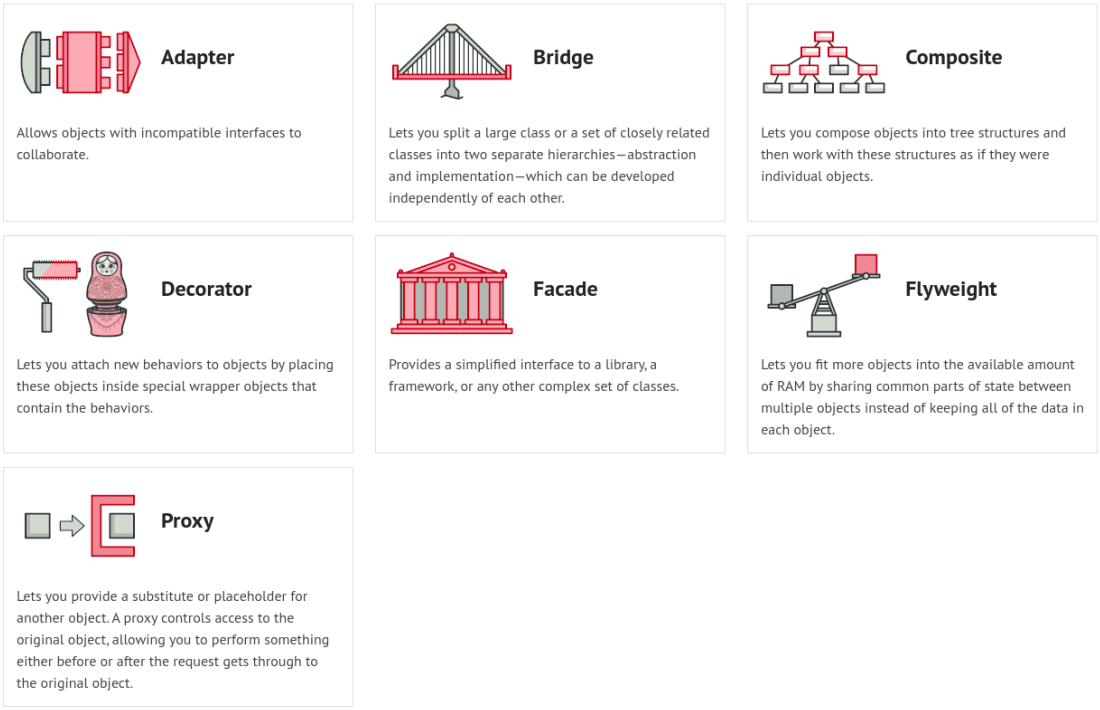
   The Singleton's constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

▼ Structural Pattern

- It focuses on composition of classes and objects, and how they can be combined to form larger structures or solve more complex problems.
- These patterns help to simplify the relationships between classes and objects, and make it easier to modify or extend the system without affecting the rest of the code.

Structural design patterns include:

1. `Adapter pattern`: This pattern allows incompatible classes to work together by creating a bridge between them.
2. `Bridge pattern:` This pattern separates the abstraction and implementation of a class, allowing them to vary independently.
3. `Composite pattern`: This pattern allows objects to be structured into tree-like hierarchies, making it easier to work with individual and groups of objects.
4. `Decorator pattern`: This pattern adds behavior to an object dynamically, without affecting the behavior of other objects.
5. `Facade pattern:` This pattern provides a simplified interface to a complex system, making it easier for clients to use.
6. `Flyweight pattern`: This pattern reduces memory usage by sharing common parts of objects between multiple instances.
7. `Proxy pattern`: This pattern provides a placeholder for an object to control access to it, such as by adding security or caching.



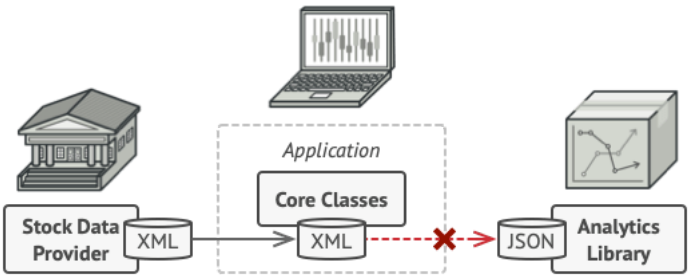
▼ Adapter Pattern

- It allows incompatible classes to work together by creating a bridge between them.
- If two or more classes have incompatible interfaces, but still want to use them together in the system.
- It creates a `adapter class` that wraps one of incompatible class and implements the interface of other class. It then translates calls from the interface of other class into calls that wrapped class can understand.

▼ Problem

Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.

At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.



You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.

You could change the library to work with XML. However, this might break some existing code that relies on the library. And worse, you might not have access to the library's source code in the first place, making this approach impossible.

```
// Third-party library with an incompatible interface
class ThirdPartyLibrary {
  constructor() {}

  methodA() {
      // ...
  }

  methodB() {
      // ...
  }
}

// Our application code that uses the third-party library directly
class Application {
  constructor() {
      this.thirdPartyLibrary = new ThirdPartyLibrary();
  }

  doSomething() {
      this.thirdPartyLibrary.methodA();
      this.thirdPartyLibrary.methodB();
  }
}

// Client code that uses our application code
class Client {
  constructor() {
      this.application = new Application();
  }

  doSomething() {
      this.application.doSomething();
  }
}

// Usage
const client = new Client();
client.doSomething();
```

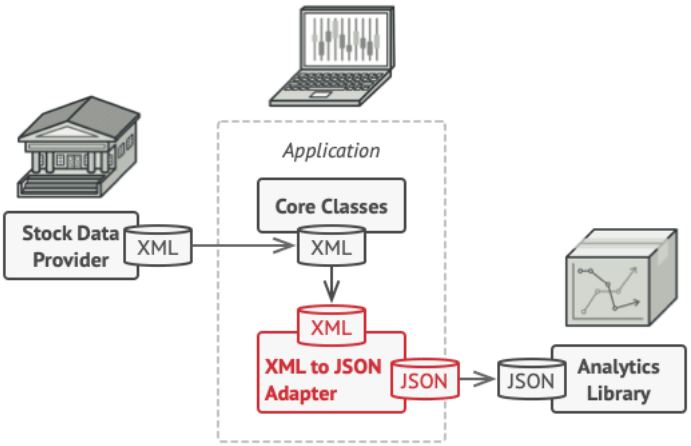
The problem with this approach is that if the third-party library changes its interface(removing or renaming methods, adding new methods that is unused, change parameter and return types), we would have to modify our application code to match the new interface. This can be time-consuming and error-prone, especially if the third-party library is used in multiple places throughout our application.

▼ Solution

Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here’s how it works:

1. The adapter gets an interface, compatible with one of the existing objects.
2. Using this interface, the existing object can safely call the adapter’s methods.
3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

Sometimes it’s even possible to create a two-way adapter that can convert the calls in both directions.



Let's get back to our stock market app. To solve the dilemma of incompatible formats, you can create XML-to-JSON adapters for every class of the analytics library that your code works with directly. Then you adjust your code to communicate with the library only via these adapters. When an adapter receives a call, it translates the incoming XML data into a JSON structure and passes the call to the appropriate methods of a wrapped analytics object.

```
// Third-party library with an incompatible interface
class ThirdPartyLibrary {
  constructor() {}

  methodA() {
      // ...
  }

  methodB() {
      // ...
  }
}

// Our application interface
class OurApplicationInterface {
  constructor() {}

  method1() {
      // ...
  }

  method2() {
      // ...
  }
}

// Adapter class that adapts the third-party library to our application interface
class ThirdPartyLibraryAdapter inherit OurApplicationInterface {
  constructor() {
      this.thirdPartyLibrary = new ThirdPartyLibrary();
  }

  method1() {
      this.thirdPartyLibrary.methodA();
  }

  method2() {
      this.thirdPartyLibrary.methodB();
  }
}

// Our application code that uses the adapter class
class Application {
  constructor() {
      this.adapter = new ThirdPartyLibraryAdapter();
  }

  doSomething() {
      this.adapter.method1();
```

```
      this.adapter.method2();
  }
}

// Client code that uses our application code
class Client {
  constructor() {
      this.application = new Application();
  }

  doSomething() {
      this.application.doSomething();
  }
}

// Usage
const client = new Client();
client.doSomething();
```

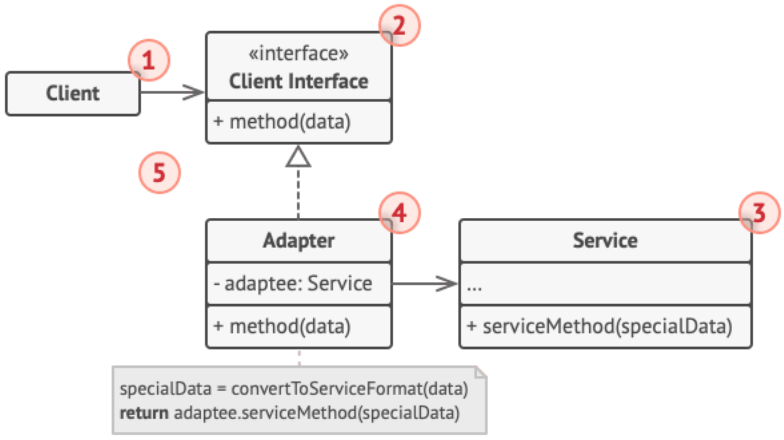
In this example, we have a third-party library with an incompatible interface (`ThirdPartyLibrary`) and our own application interface (`OurApplicationInterface`). We create an adapter class (`ThirdPartyLibraryAdapter`) that wraps the third-party library and implements our application interface. The adapter class translates the calls from our application interface into calls that the third-party library can understand.

By using the Adapter pattern, we can isolate our application code from any changes in the third-party library's interface, and minimize the impact of such changes on our code. The adapter class acts as a buffer between our application code and the third-party library, translating any changes in the third-party library's interface into a consistent interface that our application code can use.

▼ Structure

Object adapter

This implementation uses the object composition principle: the adapter implements the interface of one object and wraps the other one. It can be implemented in all popular programming languages.

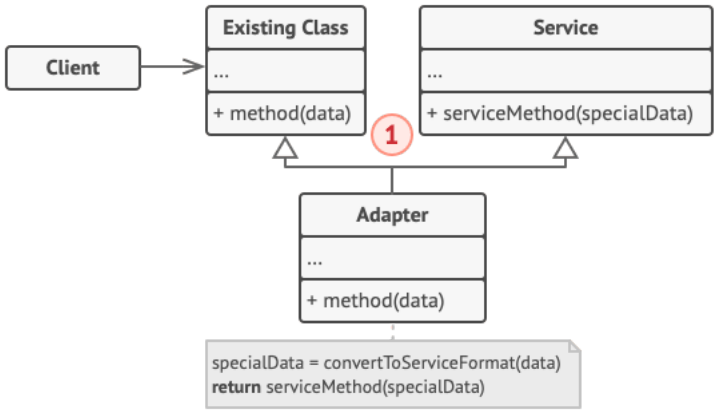


1. The **Client** is a class that contains the existing business logic of the program.
2. The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.
3. The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.
4. The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the adapter interface and translates them into calls to the wrapped service object in a format it can understand.

5. The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

Class adapter

This implementation uses inheritance: the adapter inherits interfaces from both objects at the same time. Note that this approach can only be implemented in programming languages that support multiple inheritance, such as C++.



1. The **Class Adapter** doesn't need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

▼ Bridge

▼ Intent

- Splits large class into two separate hierarchies: abstraction and implementation.
- We can create abstraction that defines interface that client code uses to interact with system and separate implementation of that interface.
- This allows to swap out different implementation of interface without affecting client code, making system more flexible and easier to maintain.

▼ Problem

```
// Without Bridge Pattern

// Web canvas implementation
class WebCanvas {
  renderSquare() {
    // ...
  }
}

// Desktop application implementation
class DesktopApp {
  renderSquare() {
    // ...
  }
}

// Square class
class Square {
  constructor(platform) {
    if (platform === 'web') {
      this.platform = new WebCanvas();
```

```
    } else if (platform === 'desktop') {
      this.platform = new DesktopApp();
    }
  }

  draw() {
    this.platform.renderSquare();
  }
}

// Usage
const mySquare = new Square('web');
mySquare.draw();
```

this example, we have two concrete implementations of the `renderSquare()` method - one for the web canvas and one for the desktop application. The `Square` class takes a `platform` parameter in its constructor and creates the appropriate implementation based on the value of the parameter. The `draw()` method of the `Square` class calls the `renderSquare()` method of the platform object.

The problem with this approach is that it tightly couples the `Square` class with the platform implementations. If we want to add support for a new platform, we need to modify the `Square` class and add a new conditional statement for the new platform. This violates the Open-Closed Principle, which states that classes should be open for extension but closed for modification.

▼ Solution

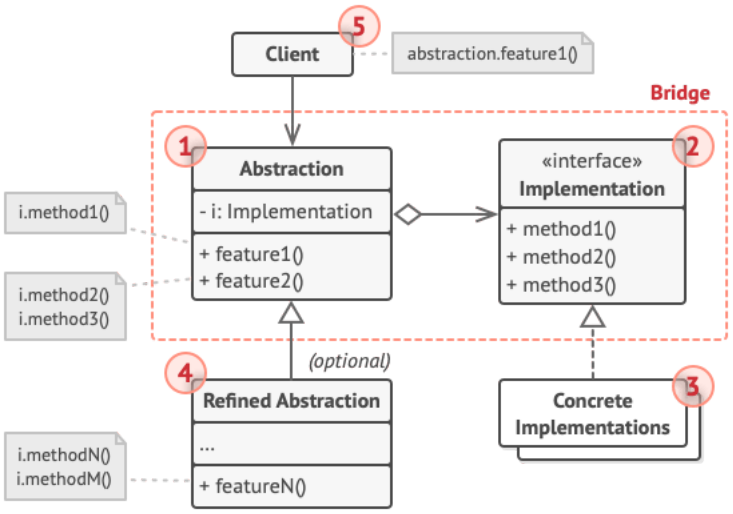
```
// With Bridge Pattern

// Platform interface
class Platform {
  renderSquare() {}
}

// Web canvas implementation of the Platform interface
class WebCanvas extends Platform {
  renderSquare() {
    // ...
  }
}

// Desktop application implementation of the Platform interface
class DesktopApp extends Platform {
  renderSquare() {
    // ...
  }
}

// Square class
class Square {
  constructor(platform) {
    this.platform = platform;
  }

  draw() {
    this.platform.renderSquare();
  }
}

// Usage
const myWebCanvas = new WebCanvas();
const mySquare = new Square(myWebCanvas);
mySquare.draw();
```

In this code, we have an abstract `Platform` interface that define the `renderSquare()` method. This allows us to define multiple implementations of the `renderSquare()` method without modifying the `Square` class. We have two concrete implementations of the `Platform` interface - one for the web canvas and one for the desktop application.

The `Square` class takes an instance of the `Platform` interface as a constructor parameter and uses it to draw the square. This allows us to vary the implementation of the `Platform` interface independently of the `Square` class. For example, we can create a new implementation of the `Platform` interface for a mobile application without modifying the `Square` class.

The advantage of this approach is that it follows the Open-Closed Principle and allows for easier extension and modification of the code. However, the downside is that it adds an additional layer of abstraction, which may increase the complexity of the code. The Bridge pattern should be used judiciously, and only when it provides real benefits over simpler alternatives.

▼ Structure



1. The **Abstraction** provides high-level control logic. It relies on the implementation object to do the actual low-level work.
2. The **Implementation** declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.

   The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.
3. **Concrete Implementations** contain platform-specific code.
4. **Refined Abstractions** provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface.
5. Usually, the **Client** is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.

▼ Composite

- It lets us compose objects into tree structure and work as if they're individual object.
- It allows client to treat individual object. Client doesn't need to know weather an object is leaf or branch. Treats all object as one.
- Has leaf and branches.
- It defines methods to add, remove and access children.
- By implementing this interface both leaves and branch can be treated same by client.

▼ Intent

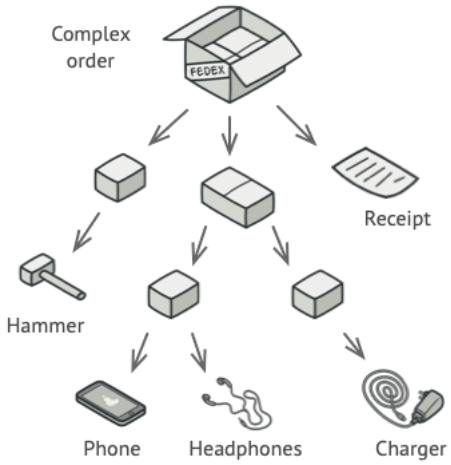
- It lets us compose objects into tree structure and work as if they're individual object.
- It allows client to treat individual object. Client doesn't need to know weather an object is leaf or branch. Treats all object as one.

▼ Problem

Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.

For example, imagine that you have two types of objects: `Products` and `Boxes`. A `Box` can contain several `Products` as well as a number of smaller `Boxes`. These little `Boxes` can also hold some `Products` or even smaller `Boxes`, and so on.

Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes. How would you determine the total price of such an order?



An order might comprise various products, packaged in boxes, which are packaged in bigger boxes and so on. The whole structure looks like an upside down tree.

You could try the direct approach: unwrap all the boxes, go over all the products and then calculate the total. That would be doable in the real world; but in a program, it's not as simple as running a loop. You have to know the classes of `Products` and `Boxes` you're going through, the nesting level of the boxes and other nasty details beforehand. All of this makes the direct approach either too awkward or even impossible.

```
class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }

  getPrice() {
    return this.price;
  }
}

class Box {
  constructor(name) {
    this.name = name;
    this.items = [];
  }

  addItem(item) {
    if (item instanceof Product) {
      this.items.push(item);
    } else {
      throw new Error("Invalid item in the box");
    }
  }

  removeItem(item) {
    const index = this.items.indexOf(item);
    if (index !== -1) {
      this.items.splice(index, 1);
    }
  }
```

```
  getPrice() {
    let totalPrice = 0;
    for (const item of this.items) {
      totalPrice += item.getPrice();
    }
    return totalPrice;
  }
}

// Example usage
const product1 = new Product("Shirt", 25);
const product2 = new Product("Shoes", 50);
const box1 = new Box("Box1");
box1.addItem(product1);
box1.addItem(product2);

// This works fine
console.log(box1.getPrice());  // Output: 75

const product3 = new Product("Pants", 30);
const box2 = new Box("Box2");
box2.addItem(product3);
try {
  box2.addItem(box1); // This will throw an error
} catch (error) {
  console.log(error.message); // Output: Invalid item in the box
}

// This won't work as expected
console.log(box2.getPrice());  // Output: 30
```

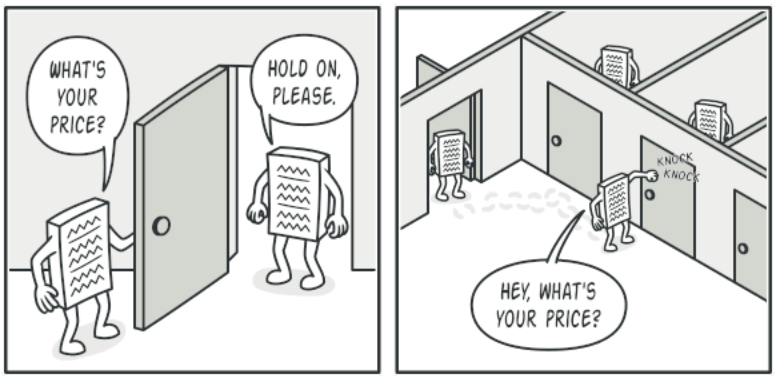
In this example, `Product` and `Box` are two separate classes, and `Box` can only contain `Product` objects, not other `Box` objects. This means that if we want to create a box that contains other boxes, we have to create a separate class for that, which can quickly become unwieldy as the complexity of the nesting increases.

As a result, this approach is not scaleable and can lead to code duplication and maintenance issues. It also makes it difficult to calculate the total price of an order that contains both products and boxes, as there is no unified way to calculate the price of a set of items.

▼ Solution

The Composite pattern suggests that you work with `Products` and `Boxes` through a common interface which declares a method for calculating the total price.

How would this method work? For a product, it'd simply return the product's price. For a box, it'd go over each item the box contains, ask its price and then return a total for this box. If one of these items were a smaller box, that box would also start going over its contents and so on, until the prices of all inner components were calculated. A box could even add some extra cost to the final price, such as packaging cost.



The Composite pattern lets you run a behavior recursively over all components of an object tree.

The greatest benefit of this approach is that you don’t need to care about the concrete classes of objects that compose the tree. You don’t need to know whether an object is a simple product or a sophisticated box. You can treat them all the same via the common interface. When you call a method, the objects themselves pass the request down the tree.

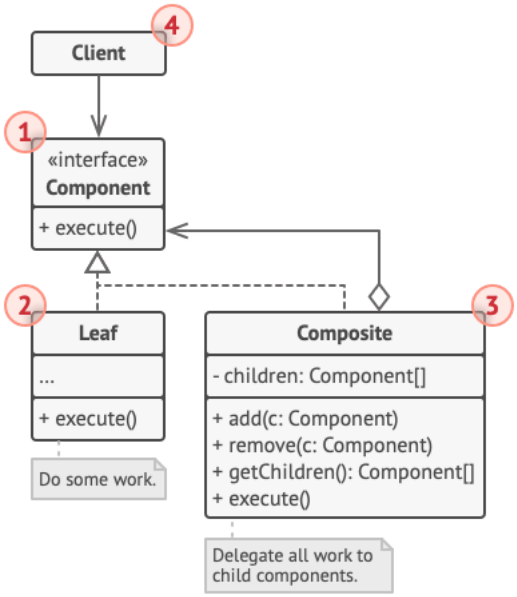
```
class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }

  getPrice() {
    return this.price;
  }
}

class Box {
  constructor(name) {
    this.name = name;
    this.items = [];
  }

  addItem(item) {
    this.items.push(item);
  }

  removeItem(item) {
    const index = this.items.indexOf(item);
    if (index !== -1) {
      this.items.splice(index, 1);
    }
  }

  getPrice() {
    let totalPrice = 0;
    for (const item of this.items) {
      totalPrice += item.getPrice();
    }
    return totalPrice;
  }
}

class Order {
  constructor() {
    this.items = [];
  }

  addItem(item) {
    this.items.push(item);
  }

  removeItem(item) {
    const index = this.items.indexOf(item);
    if (index !== -1) {
      this.items.splice(index, 1);
    }
  }

  getPrice() {
    let totalPrice = 0;
    for (const item of this.items) {
      totalPrice += item.getPrice();
    }
    return totalPrice;
  }
}

// Example usage
const product1 = new Product("Shirt", 25);
const product2 = new Product("Shoes", 50);
const box1 = new Box("Box1");
box1.addItem(product1);
box1.addItem(product2);

const product3 = new Product("Pants", 30);
const box2 = new Box("Box2");
box2.addItem(product3);
```

```
box2.addItem(box1);

const order = new Order();
order.addItem(product1);
order.addItem(box2);

console.log(order.getPrice());  // Output: 105
```

In this example, `Product` and `Box` are defined as before, but a new `Order` class is added to represent an order that can contain both `Product` and `Box` objects. The `Order` class has `addItem` and `removeItem` methods to add and remove items from its list of items, respectively. The `getPrice` method is implemented recursively to calculate the total price of all the items in the order, including any sub-boxes.

With this approach, we can easily add any number of products and boxes to an order, and the `getPrice` method will correctly calculate the total price of the order. This makes the code more scalable and easier to maintain, as we don't need to create separate classes for each level of nesting.

▼ Structure



1. The **Component** interface describes operations that are common to both simple and complex elements of the tree.
2. The **Leaf** is a basic element of a tree that doesn't have sub-elements.

   Usually, leaf components end up doing most of the real work, since they don't have anyone to delegate the work to.
3. The **Container** (aka *composite*) is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface.

   Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.
4. The **Client** works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.

▼ Decorator

- If we want to add new functionality to an object, but we don't want to modify object itself, we can create new class that inherit , but this could lead to duplication and maintainance issues. So, we use decorator.
- It allows us to add behavior to object dynamically at runtime, by wrapping it in a object of decorator class.
- Decorator class has same interface as original object, so it can be used the same way.

▼ Intent

- It lets us attach or add new behavior to objects by placing these objects inside special wrapper objects that contain the behavior.

▼ Problem

```
class Coffee {
  constructor() {
    this.cost = 2;
  }

  getCost() {
    return this.cost;
  }
}

// Problematic code:
class CoffeeWithMilk extends Coffee {
  constructor() {
    super();
    this.cost = 3; // This modifies the original object
  }
}

const coffee = new Coffee();
const coffeeWithMilk = new CoffeeWithMilk();

console.log(coffee.getCost());            // Output: 2
console.log(coffeeWithMilk.getCost());    // Output: 3
```

In this example, a `Coffee` class is defined with a `getCost` method that returns its price. Then, a new `CoffeeWithMilk` class is created by inheriting from the `Coffee` class and modifying its cost to include the cost of milk.

The problem with this approach is that it modifies the original `Coffee` object, which can lead to maintenance issues and make the code more difficult to reason about. Additionally, if we want to add more options, such as sugar or cinnamon, we would need to create new classes for each combination, which can quickly become unmanageable.

Using the Decorator pattern, we can avoid these issues by creating separate decorator classes that can be added to any `Coffee` object at runtime, without modifying its original code. This makes the code more flexible and easier to maintain, as we can add or remove decorators as needed.

▼ Solution

```
class Coffee {
  getCost() {
    return 2;
  }
}

class MilkDecorator {
  constructor(coffee) {
    this.coffee = coffee;
  }

  getCost() {
    return this.coffee.getCost() + 1;
  }
}
```

```
class SugarDecorator {
  constructor(coffee) {
    this.coffee = coffee;
  }

  getCost() {
    return this.coffee.getCost() + 0.5;
  }
}

// Create a coffee with sugar and milk
let coffee = new Coffee();
coffee = new MilkDecorator(coffee);
coffee = new SugarDecorator(coffee);

console.log(coffee.getCost());  // Output: 3.5
```

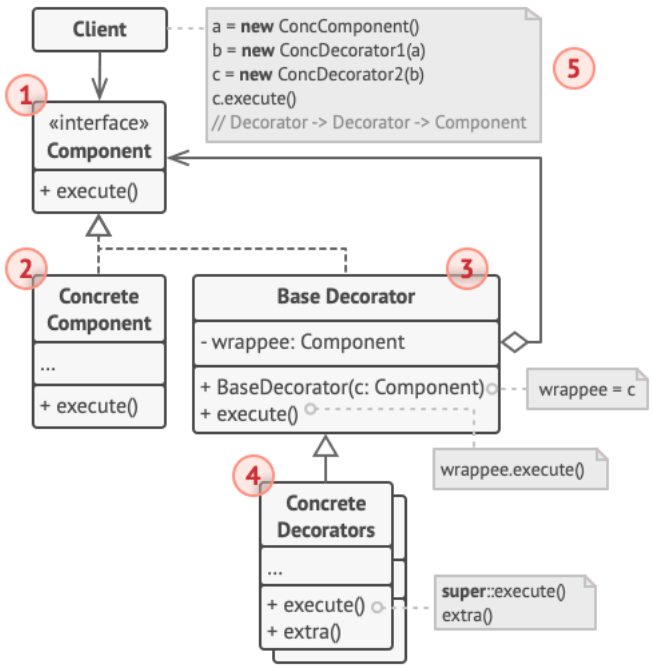
In this example, a `Coffee` class is defined with a `getCost` method that returns the base price of the coffee. Then, a `MilkDecorator` class and a `SugarDecorator` class are created that wrap a `Coffee` object and add the cost of milk and sugar to its price, respectively.

To create a coffee object with both sugar and milk, we first create a `Coffee` object, and then wrap it in a `MilkDecorator` object and a `SugarDecorator` object using the decorator pattern. The order in which we apply the decorators matters, as it affects the final price of the coffee.

Finally, we call the `getCost` method on the decorated coffee object to get its total price, which includes the cost of both sugar and milk. In this example, the total cost of the coffee is 3.5 dollars.

Using the Decorator pattern, we can easily add or remove decorators to create customized coffee objects with different combinations of ingredients, without modifying the original `Coffee` class or creating new classes for each combination.

▼ Structure



1. The **Component** declares the common interface for both wrappers and wrapped objects.
2. **Concrete Component** is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.

3. The **Base Decorator** class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.
4. **Concrete Decorators** define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.
5. The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

▼ Facade

- It is the design pattern that provides simplifies interface to a library, a framework or other complex set of class.
- It provides easy-to-use way of accessing complex system
- It acts like "front-door" to system shielding user from complexity and details of systems inner working.
- It makes software easier to use and maintain.

▼ Intent

- To provide simplified interface to a complex system, making it easier to use and understand.

▼ Problem

```
class VideoPlayer {
  constructor(video) {
    this.video = video;
  }

  play() {
    // Code to play the video
  }

  pause() {
    // Code to pause the video
  }

  stop() {
    // Code to stop the video
  }

  setVolume(volume) {
    // Code to set the volume of the video
  }

  setPlaybackSpeed(speed) {
    // Code to set the playback speed of the video
  }

  // ... and many more methods
}

// Example usage:
const video = new Video();
const player = new VideoPlayer(video);

player.play();
player.setVolume(0.5);
player.setPlaybackSpeed(1.5);
player.pause();
player.stop();
```

In this example, a `VideoPlayer` class is defined with many methods for controlling the playback of a video. Each method performs a different action, such as playing, pausing, stopping, setting the volume, and setting the playback speed.

The problem with this code is that it can be overwhelming and confusing for the user. The user needs to know which method to call for each action, and how to call them in the correct order to achieve the desired result.

Additionally, if the video player needs to support different types of video formats, or different sources for the videos, the code would become even more complex and difficult to use.

Using a Facade pattern, we can simplify the interface of the video player and make it easier to use. We could create a `VideoPlayerFacade` class that provides a simplified interface for common tasks, such as `play`, `pause`, `stop`, and `setVolume`. The Facade would handle all the complex operations behind the scenes, making it easier for the user to perform these tasks without needing to understand the internal workings of the video player.

▼ Solution

```
class VideoPlayer {
  constructor(video) {
    this.video = video;
  }

  play() {
    // Code to play the video
  }

  pause() {
    // Code to pause the video
  }

  stop() {
    // Code to stop the video
  }

  setVolume(volume) {
    // Code to set the volume of the video
  }

  setPlaybackSpeed(speed) {
    // Code to set the playback speed of the video
  }

  // ... and many more methods
}

class VideoPlayerFacade {
  constructor(videoPlayer) {
    this.videoPlayer = videoPlayer;
  }

  playVideo() {
    this.videoPlayer.play();
  }

  pauseVideo() {
    this.videoPlayer.pause();
  }

  stopVideo() {
    this.videoPlayer.stop();
  }

  setVolume(volume) {
    this.videoPlayer.setVolume(volume);
  }

  // ... and other simplified methods
}

// Example usage:
const video = new Video();
const player = new VideoPlayer(video);
const playerFacade = new VideoPlayerFacade(player);

playerFacade.playVideo();
playerFacade.setVolume(0.5);
playerFacade.pauseVideo();
playerFacade.stopVideo();
```

In this example, we have a `VideoPlayer` class with many methods for controlling the playback of a video. The `VideoPlayerFacade` class is created to provide a simplified interface to the `VideoPlayer` class. The `VideoPlayerFacade` class has methods like `playVideo`, `pauseVideo`, `stopVideo`, and `setVolume`, which map to the corresponding methods in the `VideoPlayer` class.

To use the video player with the Facade pattern, we first create a `Video` object, then a `VideoPlayer` object that takes the `Video` object as a parameter. We then create a `VideoPlayerFacade` object that takes the `VideoPlayer` object as a parameter.

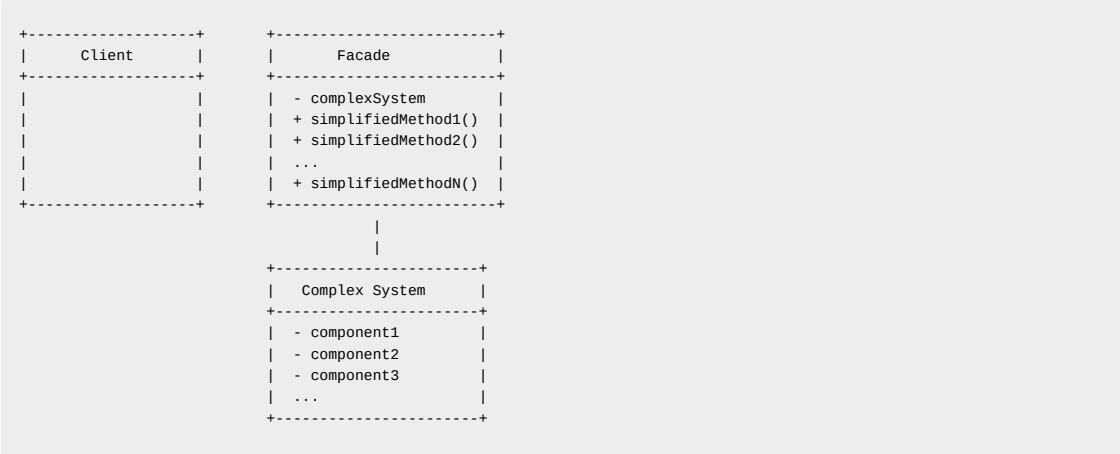
Using the Facade pattern, we can simplify the interface of the video player and make it easier to use. The user can call the simplified methods on the `VideoPlayerFacade` object, which maps to the more complex methods on the `VideoPlayer` object. This makes it easier for the user to interact with the video player without needing to understand how it works under the hood.

▼ Structure

The Facade pattern consists of the following components:

1. **Facade:** This is the main component of the pattern that provides a simplified interface to a complex system. It acts as a mediator between the user and the system, and it shields the user from the complexity and details of the system's inner workings.
2. **Complex system:** This is the system that the Facade pattern is designed to simplify. It may consist of many components, classes, or modules that work together to achieve a specific goal.
3. **Client:** This is the user of the Facade pattern who interacts with the simplified interface provided by the Facade class. The client does not need to know the details of how the complex system works under the hood.

The structure of the Facade pattern can be visualized as follows:



In this structure, the Client interacts with the Facade class, which provides a simplified interface to the Complex System. The Facade class may have many simplified methods that map to the more complex methods of the Complex System. The Complex System may consist of many components that work together to achieve a specific goal. The Facade class acts as a mediator between the Client and the Complex System, shielding the Client from the complexity and details of the system's inner workings.

▼ Fly Weight

- It lets us optimize memory usage by sharing data between multiple objects that are similar in nature.
- It reduces memory usage by sharing data between multiple object that are similar in nature.
- It helps to create shared object known as `flyweight`, that can be reused by multiple objects.
- Contains data that's common to all object that use them.

▼ Intent

- It lets us fit more objects into the available amount of RAM or memory by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

▼ Problem

```
class Character {
  constructor(char, font, size, color) {
    this.char = char;
    this.font = font;
    this.size = size;
    this.color = color;
  }

  render() {
    // Code to render the character with the given font, size, and color
  }
}

// Example usage:
const char1 = new Character('A', 'Arial', 12, 'black');
const char2 = new Character('B', 'Arial', 12, 'black');
const char3 = new Character('C', 'Arial', 12, 'black');
const char4 = new Character('A', 'Arial', 12, 'black');
```

In this example, a `Character` class is defined with properties for the character itself (`char`), the font, size, and color of the character. The class also has a `render` method to render the character with the given font, size, and color.

The problem with this code is that if you create multiple instances of the `Character` class with the same font, size, and color, the program will create a new object for each instance, even if the characters themselves are identical. This can quickly use up a lot of memory, especially if you're working with a lot of text.

For example, in the example usage code above, `char1`, `char2`, and `char3` all have different characters, but the same font, size, and color. `char4` has the same character as `char1`, but is a separate object.

Using the Flyweight pattern, we can avoid this problem by creating a shared object for each unique combination of font, size, and color, and reusing that object whenever the same combination is needed. This saves memory and improves performance.

▼ Solution

```
// flyweight class
class CharacterFlyweight {
  constructor(font, size, color) {
    this.font = font;
    this.size = size;
    this.color = color;
  }

  render(char) {
    // Code to render the character with the given font, size, and color
  }
}

// flyweight Factory
class Character {
  constructor(char, flyweight) {
    this.char = char;
    this.flyweight = flyweight;
  }

  render() {
    this.flyweight.render(this.char);
  }
}

// Example usage:
```

```
const flyweight1 = new CharacterFlyweight('Arial', 12, 'black');
const flyweight2 = new CharacterFlyweight('Times New Roman', 14, 'red');

const char1 = new Character('A', flyweight1);
const char2 = new Character('B', flyweight1);
const char3 = new Character('C', flyweight1);
const char4 = new Character('A', flyweight1);
const char5 = new Character('D', flyweight2);

char1.render();
char2.render();
char3.render();
char4.render();
char5.render();
```

In this example, we have a new `CharacterFlyweight` class, which represents the shared data for each unique combination of font, size, and color. The `Character` class now takes a `flyweight` object as a parameter instead of the font, size, and color.

When a new `Character` object is created, it uses the shared `flyweight` object for the given font, size, and color. This means that if multiple `Character` objects are created with the same font, size, and color, they will all use the same `flyweight` object.

Using the Flyweight pattern in this way can significantly reduce memory usage, especially when working with a large amount of text that contains many repeated font, size, and color combinations.

▼ Structure

The structure of the Flyweight pattern can be visualized as follows:

```
+-------------------+      +----------------------+
|       Client      |      |     FlyweightFactory  |
+-------------------+      +----------------------+
|                   |      |  - flyweights         |
|                   |      |  + getFlyweight(key)  |
|                   |      +----------------------+
|                   |                 |
|                   |                 |
|                   |      +------------------------------+
|                   |      |        Flyweight             |
|                   |      +------------------------------+
|                   |      |  - intrinsicState            |
|                   |      |  + operation(extrinsicState) |
|                   |      +------------------------------+
+-------------------+
```

In this structure, the `Client` interacts with the `FlyweightFactory` to retrieve flyweights. The `FlyweightFactory` maintains a collection of flyweights and provides a method for clients to retrieve a flyweight based on a key. The `FlyweightFactory` ensures that flyweights are shared and reused whenever possible, reducing memory usage.

The `Flyweight class` represents the shared data that is common to multiple objects. It contains an intrinsic state that is shared among all objects that use the same flyweight. It also has an operation method that takes an extrinsic state as a parameter. The operation method uses the intrinsic state and the extrinsic state to perform a task.

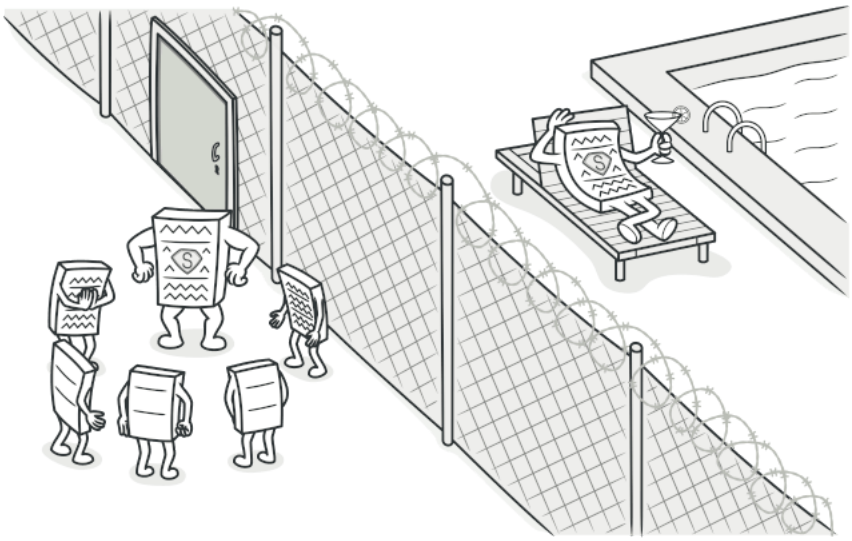
Overall, the Flyweight pattern is a way to optimize memory usage by sharing data between similar objects, which can be very useful in situations where you're working with a large amount of data and need to be mindful of memory usage.

▼ Proxy

- It allows us to control access to an object by creating a proxy that acts as intermediate between client and real object.
- Example: paid movie subscription, only paid can access premium content.
- Example: a bouncer in club.
- To control access on criteria such as user permission or authentication.
- Overall, control access to object by creating stand-in object that acts as intermediator between client and real object.

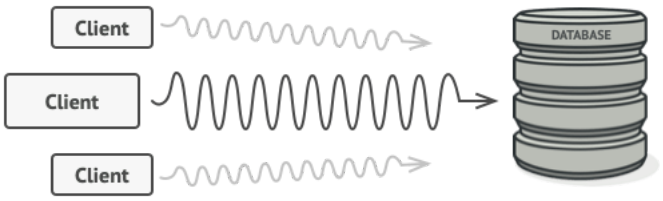
▼ Intent

- It controls the access to original object.
- The intent of the Proxy pattern is to provide a stand-in object, or "proxy," that acts as an intermediary between clients and the real object. This allows you to control access to the object and add additional functionality, without changing the interface of the object. The Proxy pattern is useful in situations where you need to control access to resources or data, or where you need to add additional functionality without modifying the real object.



▼ Problem

- suppose we have massive object that consumes vast amount of system resources. We may need it from time to time, but not always.



- You could implement lazy initialization: create this object only when it’s actually needed. All of the object’s clients would need to execute some deferred initialization code. Unfortunately, this would probably cause a lot of code duplication.

```
class Movie {
  constructor(title, isPremium) {
    this.title = title;
    this.isPremium = isPremium;
  }

  play() {
    if (this.isPremium) {
      // Code to play the premium movie
    } else {
      // Code to play the non-premium movie
    }
  }
}
```

```
// Example usage:
const movie1 = new Movie('The Matrix', false);
const movie2 = new Movie('The Godfather', true);

movie1.play(); // Plays the non-premium movie
movie2.play(); // Plays the premium movie
```

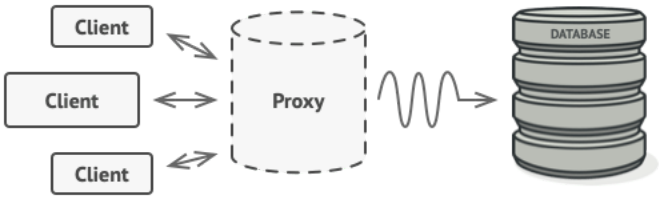
In this example, a `Movie` class is defined with a `play` method that plays the movie. The `Movie` class also has a `isPremium` flag that determines whether the movie is a premium movie or not.

The problem with this code is that the `play` method decides whether to play the premium or non-premium movie based on the `isPremium` flag. This means that any client that has access to a `Movie` object can play any movie, regardless of whether they have paid for the premium subscription.

Using the Proxy pattern, we can avoid this problem by creating a proxy object
for the premium movies that checks to see if the user has a premium subscription before allowing them to watch the movie. The proxy object acts as an intermediary between the client and the real object, and it can be used to control access to the premium content.

▼ Solution

The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.



The proxy disguises itself as a database object. It can handle lazy initialization and result caching without the client or the real database object even knowing.

But what's the benefit? If you need to execute something either before or after the primary logic of the class, the proxy lets you do this without changing that class. Since the proxy implements the same interface as the original class, it can be passed to any client that expects a real service object.

```
// real subject: client interacts with
class Movie {
  constructor(title, isPremium) {
    this.title = title;
    this.isPremium = isPremium;
  }

// subject
  play() {
    // Code to play the movie
  }
}

//proxy: has reference to movie and play() method to play
class PremiumMovieProxy {
  constructor(movie, user) {
    this.movie = movie;
    this.user = user;
  }

  play() {
    if (this.user.hasPremiumSubscription && this.movie.isPremium) {
      this.movie.play();
    } else {
      throw new Error("You don't have access to this movie");
    }
  }
}
```

```
// Example usage:
const movie1 = new Movie('The Matrix', false);
const movie2 = new Movie('The Godfather', true);

const premiumMovieProxy = new PremiumMovieProxy(movie2,
 { hasPremiumSubscription: true });

movie1.play(); // Plays the non-premium movie
premiumMovieProxy.play(); // Plays the premium movie,
// but only if the user has a premium subscription
```

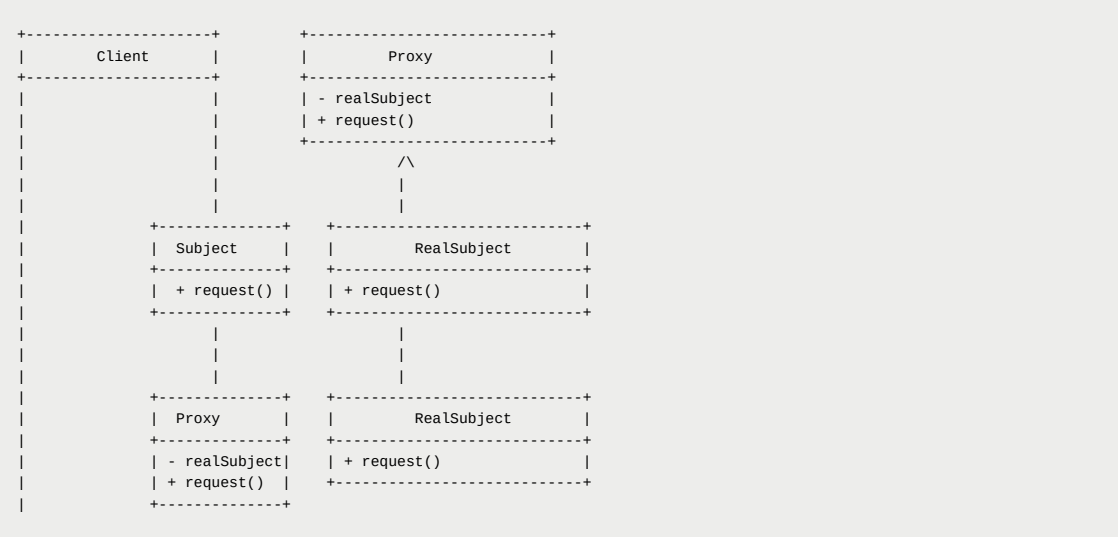
In this example, a `PremiumMovieProxy` class is defined that acts as a proxy between the client and the real `Movie` object. The `PremiumMovieProxy` class checks to see if the user has a premium subscription before allowing them to watch the movie. If the user has a premium subscription and the movie is a premium movie, the `PremiumMovieProxy` class allows the client to play the movie. If not, an error is thrown.

▼ Structure

In this structure, there are three main components:

1. **Subject:** This is the interface that defines the common methods that both the RealSubject and the Proxy implement.
2. **RealSubject:** This is the class that implements the Subject interface and represents the real object that the client wants to interact with.
3. **Proxy:** This is the class that also implements the Subject interface, but acts as a stand-in for the RealSubject. The Proxy class has a reference to the RealSubject and can add additional functionality, such as logging, caching, or security checks, before calling the RealSubject's request method.

The structure of the Proxy pattern can be visualized as follows:



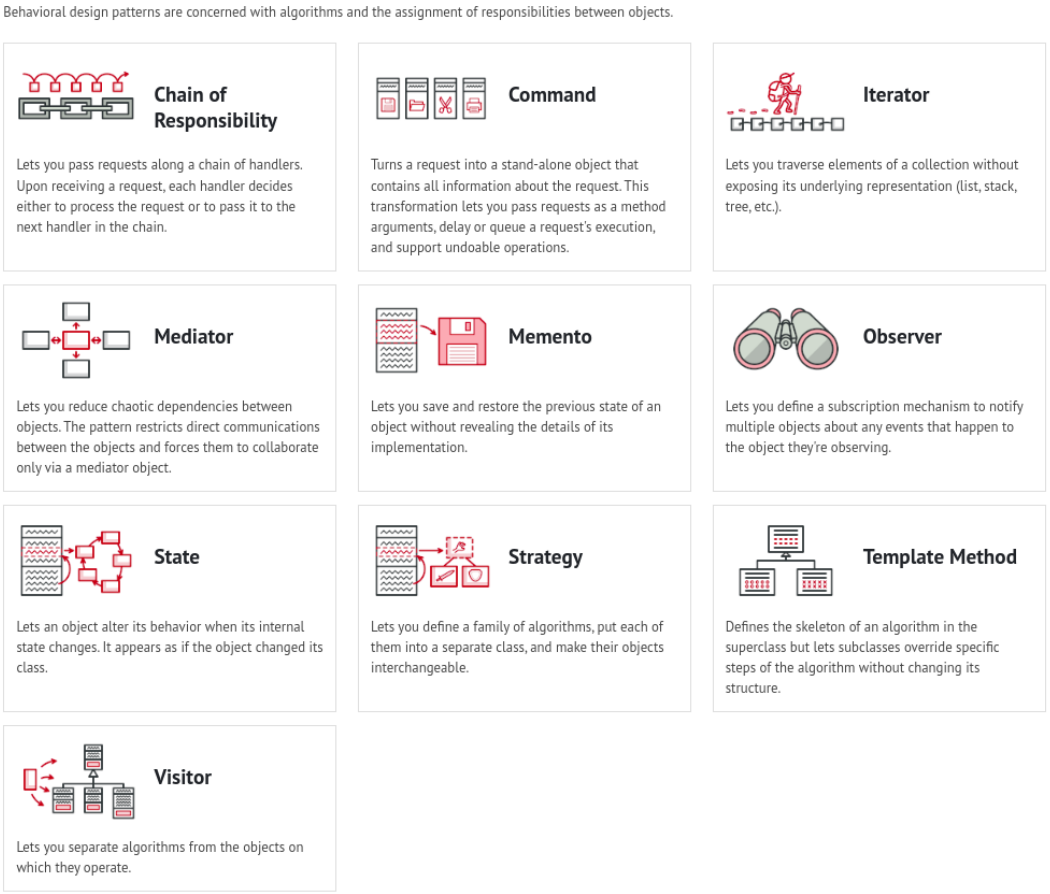
Overall, the Proxy pattern provides a way to control access to an object and add additional functionality, without changing the interface of the object. By creating a stand-in object, or "proxy," that acts as an intermediary between the client and the real object, the Proxy pattern can provide a level of indirection and allow for additional functionality to be added as needed.

▼ Behavioral Pattern

- Are concerned with algorithm and the assignment of responsibilities between them.

- It focuses on interaction and communication between objects and classes and distribution of responsibilities and behavior between them.

`Types:`



▼ Chain of Responsibility

▼ Intent

- Pass request along the chain of handlers.
- On getting a request, the handler decides either to process that request or pass it to next handler in chain.
- Client doesn't know which object will handle request. The request is passes down a chain of objects until on of them handles it.
- It helps to create modular and flexible code, objects are added or removed from chain without affecting the client code.

▼ Problem

Let's say you are a customer service representative at a call center. You receive a call from a customer who has a question about their billing statement. Instead of being able to handle the question yourself, you have to transfer the call to a manager, who then has to transfer the call to an accountant, who finally has the knowledge to answer the customer's question.

Without the Chain of Responsibility pattern, the customer would need to know exactly who to contact to get their question answered. This would require them to have a deep understanding of the company's internal structure and

who is responsible for what. Additionally, if the person they contacted was not able to help them, they would need to start the process all over again with someone else.

This can be frustrating for the customer, and it can also lead to inefficiencies within the company. With the Chain of Responsibility pattern, each person in the chain is responsible for a specific task or set of tasks, and the request is automatically passed down the chain until it is handled. This makes it easier for the customer to get their question answered, and it also allows the company to operate more efficiently by ensuring that each person is focused on their specific responsibilities.

```
class CustomerCare {
  constructor() {
    this.handlers = []
  }

  addHandler(handler) {
    this.handlers.push(handler)
  }

  handleRequest(request) {
    for (let i = 0; i < this.handlers.length; i++) {
      if (this.handlers[i].canHandleRequest(request)) {
        return this.handlers[i].handle(request)
      }
    }
    throw new Error('No handler found for request')
  }
}

class ProductSupport {
  canHandleRequest(request) {
    return request.category === 'product'
  }

  handle(request) {
    // Handle product support request
  }
}

class BillingSupport {
  canHandleRequest(request) {
    return request.category === 'billing'
  }

  handle(request) {
    // Handle billing support request
  }
}

class TechnicalSupport {
  canHandleRequest(request) {
    return request.category === 'technical'
  }

  handle(request) {
    // Handle technical support request
  }
}
```

In this code, we have a `CustomerCare` class that manages a list of handlers and provides a method to handle customer support requests. Each handler, such as `ProductSupport`, `BillingSupport`, and `TechnicalSupport`, provides its own implementation of the `canHandleRequest` and `handle` methods.

The `canHandleRequest` method determines whether or not the handler is able to handle the customer support request based on the category of the request. If the handler can handle the request, it returns `true`. If not, it returns `false`.

The `handleRequest` method is responsible for processing the customer support request. It iterates over the list of handlers and checks if each handler can handle the request by calling the `canHandleRequest` method. If a handler is found that can handle the request, it calls the `handle` method to process the request. If not, it throws an `Error`.

While this implementation works, it can become increasingly complex and difficult to maintain as new handlers are added. Additionally, if a handler needs to be changed or removed, it will require modifying the `CustomerCare` class, which can introduce bugs and make the code harder to test. Using the Chain of Responsibility design pattern can help to decouple the customer support logic from the individual handlers and make the code more modular and easier to maintain.

▼ Solution

The solution to the problem of having to transfer a call multiple times to different people until the customer's question is answered is to use the Chain of Responsibility pattern.

In the context of a call center, the Chain of Responsibility pattern could be implemented by setting up a system where calls are routed to different departments or individuals based on the nature of the customer's question. For example, calls related to billing questions could be routed to the accounting department, while calls related to technical issues could be routed to the IT department.

Each department or individual in the chain would have a specific responsibility and the ability to handle the customer's question. If they are not able to handle the question, the call would be automatically passed down the chain until it is handled by someone who can.

By using the Chain of Responsibility pattern, customers can get their questions answered more efficiently, without having to know who to contact or go through multiple transfers. Additionally, it allows each person in the chain to focus on their specific responsibilities, leading to more efficient and effective customer service.

```
class CustomerCareHandler {
  setNextHandler(handler) {
    this.nextHandler = handler;
    return handler;
  }

  handleRequest(request) {
    if (this.canHandleRequest(request)) {
      return this.handle(request);
    } else if (this.nextHandler) {
      return this.nextHandler.handleRequest(request);
    } else {
      throw new Error('No handler found for request');
    }
  }

  canHandleRequest(request) {
    throw new Error('Subclass must implement canHandleRequest method');
  }

  handle(request) {
    throw new Error('Subclass must implement handle method');
  }
}

class ProductSupportHandler extends CustomerCareHandler {
  canHandleRequest(request) {
    return request.category === 'product';
  }

  handle(request) {
    // Handle product support request
  }
}

class BillingSupportHandler extends CustomerCareHandler {
  canHandleRequest(request) {
    return request.category === 'billing';
  }

  handle(request) {
    // Handle billing support request
  }
}

class TechnicalSupportHandler extends CustomerCareHandler {
  canHandleRequest(request) {
    return request.category === 'technical';
  }
```

```
  handle(request) {
    // Handle technical support request
  }
}
```

In this example, we have a `CustomerCareHandler` abstract class that defines the basic structure of a customer care handler. Each concrete handler, such as `ProductSupportHandler`, `BillingSupportHandler`, and `TechnicalSupportHandler`, inherits from this class and provides its own implementation of the `canHandleRequest` and `handle` methods.

The `canHandleRequest` method determines whether or not the handler is able to handle the customer care request based on the category of the request. If the handler can handle the request, it returns `true`. If not, it returns `false`.

The `handle` method is responsible for processing the customer care request. It takes in the request object and performs the necessary actions to handle the request.

The `setNextHandler` method is used to set the next handler in the chain. It returns the next handler so that we can chain the handlers together.

The `handleRequest` method is responsible for processing the customer care request. It first checks if the current handler can handle the request by calling `canHandleRequest`. If it can, it calls the `handle` method to process the request. If not, it passes the request to the next handler in the chain, if there is one. If there is no handler that can handle the request, it throws an `Error`.

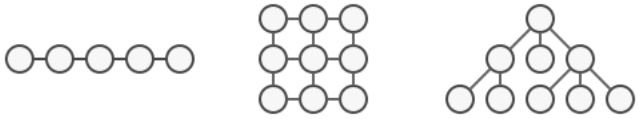
Using this implementation, we can create a chain of customer care handlers that can handle different types of customer care requests, such as product support, billing support, and technical support. This allows us to decouple the customer care logic from the individual handlers, making it more modular and easier to maintain.

▼ Iterator

- It allows us to traverse through a collection of items without having to know the underlying structure of the collection.
- It provides a way to access elements of an object sequentially without exposing its underlying representation.

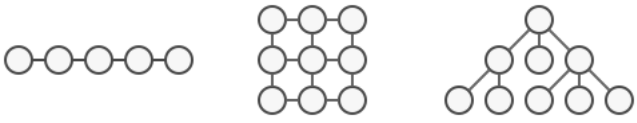
▼ Intent

- To provide a way to traverse through a collection of items without having to know the underlying structure of the collection. collection can be any data structure.



▼ Problem

Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.

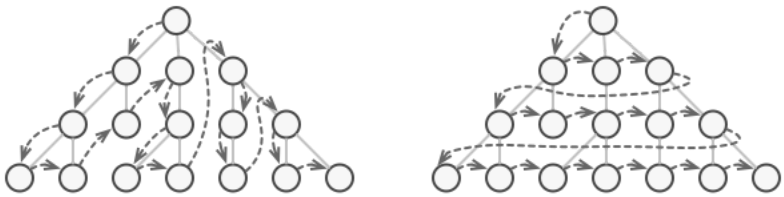


Various types of collections.

Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.

But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.

This may sound like an easy job if you have a collection based on a list. You just loop over all of the elements. But how do you sequentially traverse elements of a complex data structure, such as a tree? For example, one day you might be just fine with depth-first traversal of a tree. Yet the next day you might require breadth-first traversal. And the next week, you might need something else, like random access to the tree elements.



The same collection can be traversed in several different ways.

Adding more and more traversal algorithms to the collection gradually blurs its primary responsibility, which is efficient data storage. Additionally, some algorithms might be tailored for a specific application, so including them into a generic collection class would be weird.

On the other hand, the client code that's supposed to work with various collections may not even care how they store their elements. However, since collections all provide different ways of accessing their elements, you have no option other than to couple your code to the specific collection classes.

```
// A collection of names
const names = ['Alice', 'Bob', 'Charlie', 'Dave'];

// Traversing the collection using a for loop
for (let i = 0; i < names.length; i++) {
  console.log(names[i]);
}

// Traversing the collection using a while loop
let i = 0;
while (i < names.length) {
  console.log(names[i]);
  i++;
}

// Traversing the collection using a do-while loop
let j = 0;
do {
  console.log(names[j]);
  j++;
} while (j < names.length);
```

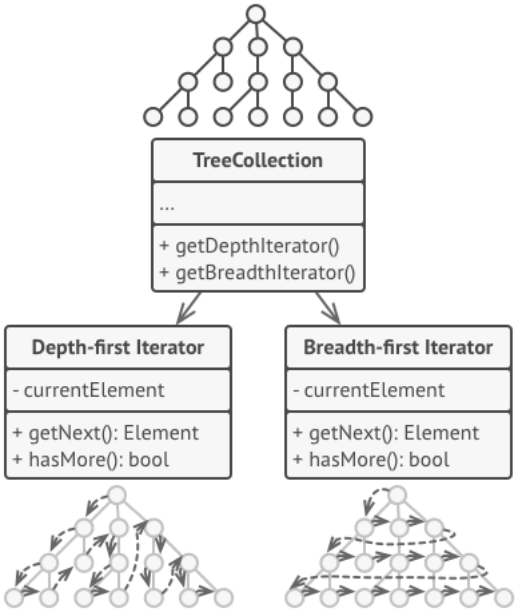
In this example, we have a collection of names stored in an array. To traverse the collection, we are using three different types of loops: for, while, and do-while. This code is problematic because it tightly couples the traversal algorithm with the collection implementation, making it difficult to switch to a different traversal algorithm or to use a different type of collection.

For example, if we wanted to switch from an array to a linked list, we would need to rewrite the entire traversal code to work with the new data structure. This could lead to code duplication, errors, and decreased maintainability.

Using the Iterator design pattern can solve this problem by providing a consistent and flexible way to traverse any type of collection, regardless of its implementation details. By abstracting the traversal logic into a separate object, we can decouple it from the collection implementation, making it easier to switch between different types of collections or traversal algorithms without affecting the rest of the code.

▼ Solution

The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an *iterator*.



Iterators implement various traversal algorithms. Several iterator objects can traverse the same collection at the same time.

In addition to implementing the algorithm itself, an iterator object encapsulates all of the traversal details, such as the current position and how many elements are left till the end. Because of this, several iterators can go through the same collection at the same time, independently of each other.

Usually, iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it doesn't return anything, which means that the iterator has traversed all of the elements.

All iterators must implement the same interface. This makes the client code compatible with any collection type or any traversal algorithm as long as there's a proper iterator. If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.

```
// Iterator interface
class Iterator {
  hasNext() {
    throw new Error('hasNext method must be implemented');
  }

  next() {
    throw new Error('next method must be implemented');
  }
}

// Concrete iterator for an array
class ArrayIterator extends Iterator {
  constructor(array) {
    super();
    this.array = array;
    this.index = 0;
  }

  hasNext() {
    return this.index < this.array.length;
  }

  next() {
    const value = this.array[this.index];
    this.index++;
    return value;
```

```
  }
}

// Concrete iterator for a linked list
class LinkedListIterator extends Iterator {
  constructor(linkedList) {
    super();
    this.linkedList = linkedList;
    this.currentNode = linkedList.head;
  }

  hasNext() {
    return this.currentNode !== null;
  }

  next() {
    const value = this.currentNode.value;
    this.currentNode = this.currentNode.next;
    return value;
  }
}

// Collection interface
class Collection {
  createIterator() {
    throw new Error('createIterator method must be implemented');
  }
}

// Concrete collection for an array
class ArrayCollection extends Collection {
  constructor(array) {
    super();
    this.array = array;
  }

  createIterator() {
    return new ArrayIterator(this.array);
  }
}

// Concrete collection for a linked list
class LinkedListCollection extends Collection {
  constructor(linkedList) {
    super();
    this.linkedList = linkedList;
  }

  createIterator() {
    return new LinkedListIterator(this.linkedList);
  }
}

// App
const arrayCollection = new ArrayCollection(['Alice', 'Bob', 'Charlie', 'Dave']);
const linkedList = new LinkedList();
linkedList.add('Alice');
linkedList.add('Bob');
linkedList.add('Charlie');
linkedList.add('Dave');
const linkedListCollection = new LinkedListCollection(linkedList);

// Traversing collections using the iterator
const arrayIterator = arrayCollection.createIterator();
while (arrayIterator.hasNext()) {
  console.log(arrayIterator.next());
}

const linkedListIterator = linkedListCollection.createIterator();
while (linkedListIterator.hasNext()) {
  console.log(linkedListIterator.next());
}
```

In this example, we have defined two iterator classes, `ArrayIterator` and `LinkedListIterator`, which provide a way to traverse an array and a linked list, respectively. We have also defined two collection classes, `ArrayCollection` and `LinkedListCollection`, which implement the `createIterator` method and return the appropriate iterator object for their respective data structure.

By using the Iterator design pattern, we have decoupled the traversal logic from the collection implementation, making it easier to switch between different types of collections or traversal algorithms without affecting the rest of the code. We can simply create a new collection object and call its `createIterator` method to obtain an iterator instance, which we can use to traverse the collection using a while loop.

▼ Mediator

- It reduces chaotic dependencies between objects.
- It restricts direct communication between objects and forces them to collaborate only via mediator object.

▼ Intent

- The mediator object acts as an intermediary between the objects and encapsulates the communication logic, allowing the objects to interact with each other without knowing each other's identities or details about their behavior.
- The Mediator pattern helps to reduce the complexity of the communication between objects by eliminating the need for direct references between them. Instead, each object communicates with the mediator, which then relays messages to the appropriate objects. This makes the code easier to maintain and extend, because changes to one object do not require changes to the others, and new objects can be added without changing the existing ones.

▼ Problem

```
import React, { useState } from "react";

function ProfileDialog() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [hasDog, setHasDog] = useState(false);
  const [dogName, setDogName] = useState("");
  const [formValid, setFormValid] = useState(false);

  const handleNameChange = (event) => {
    setName(event.target.value);
    validateForm();
  };

  const handleEmailChange = (event) => {
    setEmail(event.target.value);
    validateForm();
  };

  const handleHasDogChange = (event) => {
    setHasDog(event.target.checked);
    if (event.target.checked) {
      document.getElementById("dog-name-field").style.display = "block";
    } else {
      document.getElementById("dog-name-field").style.display = "none";
    }
    validateForm();
  };

  const handleDogNameChange = (event) => {
    setDogName(event.target.value);
    validateForm();
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    // Save the profile data
  };

  const validateForm = () => {
    const nameValid = name.trim() !== "";
    const emailValid = email.trim() !== "";
    const dogNameValid = !hasDog || (hasDog && dogName.trim() !== "");
    setFormValid(nameValid && emailValid && dogNameValid);
  };

  return (
    <form onSubmit={handleSubmit}>
```

```
      <label>
        Name:
        <input type="text" value={name} onChange={handleNameChange} />
      </label>
      <br />
      <label>
        Email:
        <input type="email" value={email} onChange={handleEmailChange} />
      </label>
      <br />
      <label>
        <input type="checkbox" checked={hasDog} onChange={handleHasDogChange} />
        I have a dog
      </label>
      <label id="dog-name-field" style={{ display: "none" }}>
        Dog's name:
        <input type="text" value={dogName} onChange={handleDogNameChange} />
      </label>
      <br />
      <button type="submit" disabled={!formValid}>
        Save
      </button>
    </form>
  );
}

export default ProfileDialog;
```

In this example, the UI elements (text fields, checkbox, submit button) are each responsible for their own state, just like in the previous example. However, instead of interacting with each other through a mediator, the `handleHasDogChange` function directly manipulates the DOM to show/hide the dog name text field.

This approach is problematic for a few reasons. First, it tightly couples the UI elements to the DOM, which makes it harder to reuse these elements in other forms of the app. Second, it makes the code harder to maintain, because any changes to the UI structure or styling could break the logic that manipulates the DOM. Finally, it violates the Separation of Concerns principle, because the UI elements are responsible for both managing their own state and manipulating the DOM, which are two separate concerns that should be handled independently.

Using the mediator pattern can help solve these problems by introducing a separate mediator component that manages the interactions between the UI elements and handles the DOM manipulation. This allows the UI elements to focus on managing their own state, and makes the code more modular and easier to maintain.

▼ Solution

```
import React, { useState } from "react";

function ProfileDialog() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [hasDog, setHasDog] = useState(false);
  const [dogName, setDogName] = useState("");
  const [formValid, setFormValid] = useState(false);

  const handleNameChange = (event) => {
    setName(event.target.value);
    validateForm();
  };

  const handleEmailChange = (event) => {
    setEmail(event.target.value);
    validateForm();
  };

  const handleHasDogChange = (event) => {
    setHasDog(event.target.checked);
  };

  const handleDogNameChange = (event) => {
    setDogName(event.target.value);
    validateForm();
  };

  const handleSubmit = (event) => {
    event.preventDefault();
```

```
    // Save the profile data
  };

  const validateForm = () => {
    const nameValid = name.trim() !== "";
    const emailValid = email.trim() !== "";
    const dogNameValid = !hasDog || (hasDog && dogName.trim() !== "");
    setFormValid(nameValid && emailValid && dogNameValid);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" value={name} onChange={handleNameChange} />
      </label>
      <br />
      <label>
        Email:
        <input type="email" value={email} onChange={handleEmailChange} />
      </label>
      <br />
      <label>
        <input type="checkbox" checked={hasDog} onChange={handleHasDogChange} />
        I have a dog
      </label>
      {hasDog && (
        <label>
          Dog's name:
          <input type="text" value={dogName} onChange={handleDogNameChange} />
        </label>
      )}
      <br />
      <button type="submit" disabled={!formValid}>
        Save
      </button>
    </form>
  );
}

export default ProfileDialog;
```

In this example, the UI elements (text fields, checkbox, submit button) are each responsible for their own state. However, instead of interacting with each other directly, they call functions that update the state of the component. The `validateForm` function is called whenever any of the form fields change, and it updates the state of the `formValid` variable, which is used to disable/enable the submit button.

The `hasDog` state variable is used to show/hide the dog name text field when the "I have a dog" checkbox is checked/unchecked. By managing the interactions between the UI elements through state and functions, we avoid tight coupling between these elements and make them easier to reuse in other forms of the app.

▼ Memento

- It is a pattern that helps to manage state of an object in a way that allows it to be restored to a previous state if needed.
- It lets us save and restore previous state of object without revealing details of its implementation.
- For example: Like in game, we can continue from the last state without starting from the beginning itself.
- It is useful in situations where we need to undo an action or revert to a previous state of object.

▼ Intent

- It lets us save and restore to the previous state of an object without revealing the details of its implementation.

▼ Problem

```
public class TextEditor {
    private String text = "";

    public void addText(String newText) {
```

```
        text += newText;
    }

    public void undo() {
        // This method attempts to undo the last operation
        // by removing the last character from the text.
        // However, it doesn't keep track of the previous state
        // of the text, so it may not work correctly in all cases.
        if (text.length() > 0) {
            text = text.substring(0, text.length() - 1);
        }
    }

    public void printText() {
        System.out.println(text);
    }
}
```

In this example, the `TextEditor` class has a `text` field that stores the current text, and methods for adding new text, undoing the last operation, and printing the text to the console. However, the `undo` method doesn't keep track of the previous state of the text, and simply removes the last character from the current text. This approach may not work correctly in all cases, especially if the text has been modified in other ways.

A better solution would be to use the Memento pattern to store the previous state of the text. Here's an example of how that could be done:

▼ Solution

```
import java.util.Stack;

public class TextEditor {
    private String text = "";
    private Stack<String> textHistory = new Stack<>();

    public void addText(String newText) {
        textHistory.push(text);
        text += newText;
    }

    public void undo() {
        if (!textHistory.isEmpty()) {
            text = textHistory.pop();
        }
    }

    public void printText() {
        System.out.println(text);
    }
}
```

In this updated version of the `TextEditor` class, the `addText` method now pushes the previous state of the text onto a stack before adding the new text. The `undo` method pops the previous state from the stack and restores the text to that state. This approach allows the user to undo any previous changes to the text, without having to keep a copy of the entire text each time.

By using the Memento pattern, we've improved the flexibility and maintainability of the code, and made it easier to extend the functionality of the text editor in the future.

▼ Observer

- It allows one object to be notifies when another object changes its state.
- It allows an object (the subject) to notify other object (the observer) when its state changes.
- It is useful when we have multiple objects that needs to know about the changes to state of another object but have no tight coupling between them.

- Defines one-to-many relation between subject and observer respectively.
    - Subject maintains list of all observer and notifies them when its state changes.
    - Observer can then update their own state or perform other actions based on new state of subject.

▼ Code quality and standards with ESLint

Javascript has very flexible syntax by default. We can change type of a variable, use single and double quote for strings, have any number of space for indentation, omit the trailing semicolons,etc.

But if we work in a team, we should have some coding standards in the same team, so that the code can be easier to review and maintain. It can make our code more professional as well.

ESLint is a great tool for this purpose.

▼ ESLint

- It is a static code analysis tool that can be used to check the code quality and coding styles of Javascript code.
- It analyzes code for potential errors, styling issues and coding standards. By using ESLint, developers can ensure their code meets certain quality and style standards, which can improve readability, maintainability and overall code quality.
- It provides different built-in rules that can be customized to fit the needs of a specific project or development team. It covers issues such as variable naming conventions, indentation and the use of specific language features. Developers can also create their own custom rules to enforce project-specific standards.
- ESLint works by parsing your code and applying a set of rules that you can configure or customize. These rules can be based on best practices, coding standards, or your own preferences. ESLint can also fix some of the problems automatically, or suggest solutions for you to apply. You can run ESLint from the command line, or integrate it with your editor, IDE, or build tool. ESLint can also work with different plugins, parsers, and formatters to support various features and frameworks, such as TypeScript, React, Vue, Prettier, and more.
- To use ESLint in a project, developers typically install it as a devDependency using npm or another package manager. They can then configure the tool to use a set of rules by creating an ESLint configuration file (usually named `.eslintrc.json`), which specifies the rules to be used and any additional configuration options.

▼ Why quality and consistent code matters?

- It can affect our productivity, performance, and security. By using ESLint, we can ensure that our code follows consistent and clear conventions. This can make our code easier to understand, debug, refactor and test. It can also prevent some bugs that may compromise our functionality and user experience.
- Code consistency ensures that our code is compatible and interoperable with other code and tools. It can affect our collaboration, integration and scalability. By ensuring that our code follows same format, structure and logic across our files, modules and components, this makes our code easier to reuse, share and extend. It also prevent from conflicts and issues that can arise from different coding styles and approaches.

▼ Getting started with ESLint.

1. `We can install ESLint`: install it globally using npm by running following command;

```
npm install -g eslint
```

or install it locally in our project directory.

```
npm install eslint --save-dev
```

2. `Create a ESLint Configuration File:` ESLint needs to be configured to work properly with our project. We need to create configuration file by running the following command in our terminal:

```
eslint --init
```

3. `Run ESLint on your code:` You can run ESLint on your code by running the following command in your terminal:

```
eslint yourfile.js
```

This command will analyze your code and report any issues that violate the rules specified in your configuration file.

4. `Fix ESLint issues:` ESLint provides suggestions for fixing issues it finds in your code. You can fix them manually or use the `-fix` option to automatically fix some of them. For example:

```
eslint --fix yourfile.js
```

This command will automatically fix some of the issues in your code based on the rules specified in your configuration file.

5. **`Integrate ESLint with our editor:`**

   Integrating ESLint with Visual Studio Code is a straightforward process. Here are the steps:

   1. `Install the ESLint extension:` Open Visual Studio Code and navigate to the Extensions view (Ctrl+Shift+X). Search for "ESLint" and install the extension by Dirk Baeumer.
   2. `Install ESLint in your project:` If you haven't already done so, install ESLint in your project by running the following command in your terminal:

      ```
      npm install eslint --save-dev
      ```

   3. `Create an ESLint configuration file:` Create an `.eslintrc` file in the root of your project directory. You can use the `eslint --init` command to create a configuration file, as I explained in my previous answer.
   4. `Configure Visual Studio Code to use ESLint:` Open your user or workspace settings in Visual Studio Code (File > Preferences > Settings or Ctrl+,). Search for "eslint.autoFixOnSave" and check the box to enable automatic fixing of ESLint issues on save. Also, search for "eslint.validate" and add the following configuration:

      ```
      "eslint.validate": [
        "javascript",
        "javascriptreact",
        "typescript",
        "typescriptreact"
      ]
      ```

      This configuration tells Visual Studio Code to validate JavaScript and TypeScript files using ESLint.

      1. Reload Visual Studio Code: After making changes to your settings, reload Visual Studio Code to apply the changes.

      That's it! Now, when you open a JavaScript or TypeScript file in Visual Studio Code, ESLint will automatically validate your code and highlight any issues it finds. You can also use the "ESLint: Fix all auto-fixable Problems" command (Ctrl+Shift+P) to automatically fix issues in your code.

▼ Customizing ESLint.

One of the great features of ESLint is that it is highly customizable and flexible. You can tailor ESLint to suit your needs and preferences by modifying the configuration file or using the command line options. You can also add or remove rules, change the severity or options of the rules, or create your own custom rules. You can also use different plugins, parsers, and formatters to enable or enhance the support for various features and frameworks. You can also use comments or directives in your code to disable or enable specific rules for certain lines or blocks.

ESLint can be customized to fit the needs of your specific project or development team. Here are some ways to customize ESLint:

1. `Adding or modifying rules:` You can add or modify ESLint rules to match your project's coding standards. Rules can be added or modified in the `.eslintrc` file using the "rules" property. For example, to enforce the use of double quotes for string literals, you can add the following rule:

```
"rules": {
  "quotes": ["error", "double"]
}
```

2. `Using plugins:` ESLint plugins provide additional rules and functionality. To use a plugin, install it using npm and add it to the "plugins" array in your `.eslintrc` file. For example, to use the `eslint-plugin-react` plugin, install it using the following command:

```
npm install eslint-plugin-react --save-dev
```

Then, add it to your `.eslintrc` file as follows:

```
"plugins": [
  "react"
]
```

You can then use the plugin's rules in your configuration file.

3. `Specifying environments:` ESLint can be configured to recognize specific environments, such as browsers, Node.js, or CommonJS. To specify environments, use the "env" property in your `.eslintrc` file. For example, to specify that your code will run in a browser environment, add the following configuration:

Copy

```
"env": {
  "browser": true
}
```

4. `Using shareable configurations:` Shareable configurations allow you to reuse configuration across multiple projects or teams. A shareable configuration is an npm package that exports an ESLint configuration object. To use a shareable configuration, install it using npm and add it to the "extends" array in your `.eslintrc` file. For example, to use the `eslint-config-airbnb` shareable configuration, install it using the following command:

```
npm install eslint-config-airbnb --save-dev
```

Then, add it to your `.eslintrc` file as follows:

```
"extends": [
  "airbnb"
]
```

These are just a few ways to customize ESLint. For more information, consult the ESLint documentation or the documentation of specific plugins

▼ Twelve-Factor App

- Twelve Factor app is a methodology for building software-as-a-service(Saas) apps that:
  - Use declarative format for setup automation, to minimize time and cost for new developers joining the project.
  - Have a clean contract with underlying os, offering maximum portability between execution environments.
  - Minimize divergence between development and production, enabling continuous deployment for maximum agility.
  - scale up without significant changes to tooling, architecture, or development practices.

The 12-factor app is a methodology for building modern, scalable, and maintainable software-as-a-service (SaaS) applications. It was first introduced by Heroku, a cloud platform provider, in 2011.

The 12 factors are a set of best practices that aim to provide a standardized approach to building cloud-native applications that are highly scalable, resilient, and easy to maintain. Here are the 12 factors:

1. `Codebase:` One codebase tracked in version control, many deploys
2. `Dependencies:` Explicitly declare and isolate dependencies
3. `Config:` Store configuration in the environment
4. `Backing services:` Treat backing services as attached resources
5. `Build, release, run:` Strictly separate build and run stages
6. `Processes:` Execute the app as one or more stateless processes
7. `Port binding:` Export services via port binding
8. `Concurrency:` Scale out via the process model
9. `Disposability:` Maximize robustness with fast startup and graceful shutdown
10. `Dev/prod parity:` Keep development, staging, and production as similar as possible
11. `Logs:` Treat logs as event streams
12. `Admin processes:` Run admin/management tasks as one-off processes

By following the 12-factor methodology, software developers can improve their ability to deliver highly scalable, maintainable, and portable applications that can run in any cloud environment.

1. `Codebase:`

- One codebase tracked in version control, many deploys.
- This factor emphasizes the importance of having a single codebase that is tracked in a version control system (such as Git) and can be deployed to multiple environments. This approach ensures that everyone is working on the same codebase and makes it easier to manage changes and track issues.

2. `Dependencies:`

- Explicitly declare and isolate dependencies
- This factor highlights the need to declare all dependencies explicitly and keep them isolated from the application code. This approach ensures that the application has all the necessary dependencies to run correctly and avoids conflicts with other dependencies.

3. `Config:`

- Store configuration in the environment
- This factor emphasizes the importance of storing configuration in the environment rather than hard-coding it into the application code. This approach makes it easier to manage configuration changes and allows for different configurations for different environments.
- A web application needs to connect to a database. The database connection string is stored as an environment variable, rather than being hard-coded in the application code. This approach allows for different configurations for different environments, such as development, staging, and production.

4. `Backing services:`

- Treat backing services as attached resources
- This factor emphasizes the importance of treating backing services (such as databases, message queues, and caches) as attached resources that can be easily swapped out or scaled independently of the application. This approach ensures that the application is decoupled from its dependencies and can be easily scaled as needed.
- example: A web application uses a cloud-based database service like Amazon RDS. The database is treated as an attached resource that can be easily scaled up or down as needed. The application code is decoupled from the database and can be easily migrated to a different database service if needed.

5. `Build,release,run:`

- Strictly separate build and run stages
- This factor emphasizes the importance of separating the build, release, and run stages of the application lifecycle. This approach ensures that the application is built and tested in a consistent and reproducible manner and can be easily deployed to different environments.
- Example: A team of developers uses a CI/CD pipeline to build and test the application, and then deploys it to a staging environment for further testing. Once the application has been tested and approved, it is deployed to the production environment. The build, release, and run stages are strictly separated to ensure that the application is built and tested in a consistent and reproducible manner.

6. `Processes:`

- Execute the app as one or more stateless processes
- This factor emphasizes the importance of executing the application as one or more stateless processes. This approach ensures that the application can be easily scaled horizontally and that any failed processes can be easily replaced.
- A web application is deployed to multiple instances running on different servers. Each instance is a stateless process that communicates with a load balancer. If an instance fails, the load balancer redirects traffic to a healthy instance. This approach ensures that the application is easily scalable and can handle a high volume of traffic.

7. `Port binding:`

- Export services via port binding
- This factor emphasizes the importance of exporting services via port binding. This approach ensures that the application is easily accessible via a network and can be easily scaled horizontally by adding more instances.
- Example: A web application exposes its functionality through an API that is accessible via a specific port. The port is bound to the application so that incoming requests are routed to the correct process or instance. This approach ensures that the application is easily accessible via a network and can be easily scaled horizontally by adding more instances.

8. `Concurrency:`

- Scale out via the process model
- This factor emphasizes the importance of scaling out the application via the process model. This approach ensures that the application can be easily scaled horizontally by adding more processes or instances.
- Example: A web application is deployed to multiple instances running on different servers. Each instance runs multiple processes, and each process handles a specific task or request. If the application needs to handle more requests, additional processes can be added to each instance. This approach ensures that the application can be easily scaled horizontally by adding more processes or instances.

9. `Disposability:`

- Maximize robustness with fast startup and graceful shutdown
- This factor emphasizes the importance of maximizing the application's robustness by ensuring that it starts up quickly and shuts down gracefully. This approach ensures that the application can be easily restarted or scaled up/down without causing disruptions.
- Example: A web application is designed to start up quickly and shut down gracefully. If an instance fails, a new instance can be quickly started up to replace it. This approach ensures that the application can be easily restarted or scaled up/down without causing disruptions.

10. `Dev/prod parity:`

- Keep development, staging, and production as similar as possible
- This factor emphasizes the importance of keeping development, staging, and production environments as similar as possible. This approach ensures that the application behaves consistently across different environments and reduces the risk of deployment issues.
- Example: A web application is developed and tested in a development environment that is as similar as possible to the production environment. The same tools, configurations, and dependencies are used in both environments to reduce the risk of deployment issues.

11. `Logs:`

- Treat logs as event streams
- This factor emphasizes the importance of treating logs as event streams that can be easily collected, stored, and analyzed. This approach ensures that the application's behavior can be easily monitored and debugged.
- Example: A web application logs all events and errors to a centralized logging system. The logs are treated as event streams that can be easily collected, stored, and analyzed. This approach allows the developers to monitor the application's behavior and debug issues
  quickly.

12. `Admin processes:`

- Run admin/management tasks as one-off processes
- This factor emphasizes the importance of running admin/management tasks as one-off processes that can be easily executed and then terminated. This approach ensures that the application's resources are not tied up by long-running administrative tasks and that these tasks can be easily repeated if needed.
- Example: A web application needs to perform administrative tasks like database migrations or backups. The tasks are performed as one-off processes that can be easily executed and then terminated. This approach ensures that the application's resources are not tied up by long-running administrative tasks and that these tasks can be easily repeated if needed.