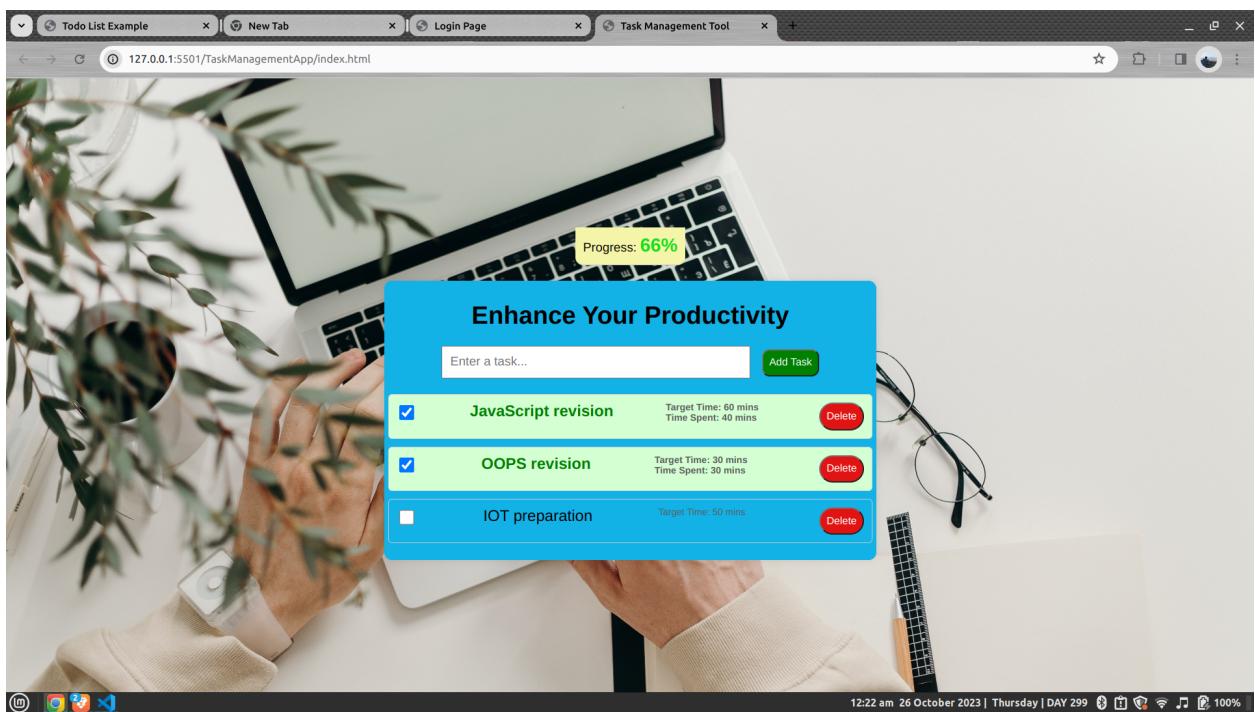
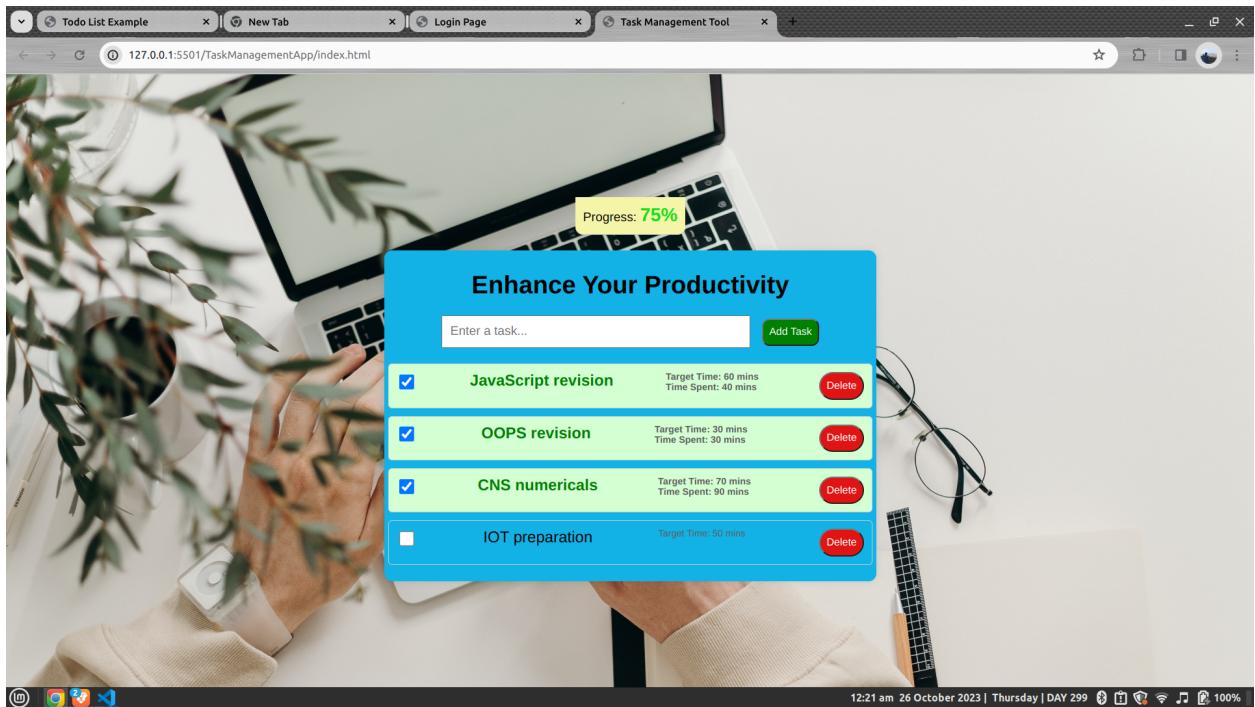


Task Management Tool

The image displays two screenshots of a web-based Task Management Tool, showing its interface and a modal dialog.

Screenshot 1: The main application window shows a blue header bar with the text "Enhance Your Productivity". Below it is a search bar containing the placeholder "Enter a task...". To the right of the search bar is a green "Add Task" button. A yellow progress bar at the top right indicates "Progress: 0%". The background features a photograph of a person's hands working on a laptop keyboard, with a potted plant on the left and a pair of glasses on the right.

Screenshot 2: This screenshot shows a modal dialog box from the application. The dialog has a white background and a thin black border. It contains the text "127.0.0.1:5501 says" and "Enter target time for this task(in minutes):" followed by an empty input field. At the bottom of the dialog are two buttons: "Cancel" and "OK". The background of this screenshot is identical to the first one, showing the same workspace scene.



HTML Code :

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Task Management Tool</title>
<link rel="stylesheet" href="1.css">
</head>
<body>
<div id="progress">Progress: <span id="progressValue">0%</span></div>
<div class="container">
<h1>Enhance Your Productivity</h1>
<input type="text" id="taskInput" placeholder="Enter a task...">
<button id="addTask">Add Task</button>
<ul id="taskList"></ul>
</div>

<script src="1.js"></script>
</body>
</html>
```

CSS CODE:

```
body {
background-image: url('bgimageProductivityApp.jpg');
background-size: cover;
background-repeat: no-repeat;
font-family: Arial, sans-serif;
background-color: cornflowerblue;
display: flex;
justify-content: center;
align-items: center;
height: 100vh;
margin: 0;
```

```
flex-direction: column; /* Stack elements vertically on smaller screens */
}

#progress {
text-align: center; /*Center the text for all screens */
padding: 10px;
background-color: rgb(245, 245, 169);
border-bottom: 1px solid #ccc;
border-radius: 0 0 10px 10px;
margin-top: 20px; /* this adds space between #progress and .container */
}

#progressValue {
font-weight: bold;
color: #12e119;
font-size: 1.5rem; /* Increase font size for better visibility on smaller
screens */
}

.container {
background-color: rgb(18, 177, 230);
padding: 5px;
border-radius: 10px;
box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
max-width: 620px;
width: 90%; /* this reduces the width slightly to fit better on smaller
screens */
text-align: center;
margin-top: 20px; /* Add space between #progress and .container */
}

h1 {
margin-bottom: 20px;
}

input[type="text"] {
font-size: medium;
padding: 10px;
```

```
width: 60%;  
max-width: 100%; /* Prevent the input from overflowing on smaller screens */  
}  
  
  
input[type="checkbox"] {  
width: 20px;  
height:20px; /* size for check box...later added*/  
}  
  
  
label {  
font-size: 20px; /*textcontent in textitem(li)..later added */  
}  
  
  
.taskDetails {  
/* display: flex; */  
justify-content: space-between;  
font-size: 75%;  
color: rgb(96, 93, 93);  
/* margin-top: 5px; */  
}  
  
  
/* .taskDetails span {  
font-weight: bold;  
color: #333;  
} */  
  
  
button {  
padding: 8px 8px;  
margin-left: 10px;  
}  
  
  
#addTask{ /* later added */  
color: rgb(243, 249, 255) ;  
border-radius: 10px;  
background-color: green;  
}  
  
  
.deleteTask{ /* later added */
```

```
color: white ;
border-radius: 20px;
background-color: rgb(225, 20, 20);
}

ul {
list-style: none;
padding: 0;
margin-top: 20px;
text-align: left;
}

li {
display: flex;
justify-content: space-between;
padding: 10px;
margin-bottom: 10px;
border: 1px solid #ccc;
border-radius: 5px;
}

li.completed {
background-color: #d3ffd3;
color: green;
font-weight: bold;
}

/* #progressValue {
font-weight: bold;
color: #12e119;
font-size: 1.5rem; Increase font size for better visibility on smaller
screens
} */
```

JAVASCRIPT CODE:

```
// Getting references to HTML elements
const taskInput = document.getElementById('taskInput');
const taskList = document.getElementById('taskList');
const addTaskButton = document.getElementById('addTask');
const progressValue = document.getElementById('progressValue');

// Variables to keep track of completed and total tasks
let completedTasks = 0;
let totalTasks = 0;

// Adding event listeners to buttons
addTaskButton.addEventListener('click', addTask);
taskList.addEventListener('click', handleTaskActions);

// Function to add a new task
function addTask() {
  const taskText = taskInput.value; // Get the task description from the input field
  const taskTime = parseFloat(prompt('Enter target time for this task(in minutes):')); // Get the target time for the task from the user

  // Check if the input is valid
  if (taskText.trim() !== '' && !isNaN(taskTime) && taskTime > 0) {
    // Create a new list item for the task
    const taskItem = document.createElement('li');
    // Create a checkbox and a label for the task description
    const checkbox = document.createElement('input');
    checkbox.type = 'checkbox';
    const label = document.createElement('label');
    label.textContent = taskText;

    // Add the checkbox and label to the task item
    taskItem.appendChild(checkbox);
    taskItem.appendChild(label);
  }
}
```

```
// Create a section for task details (target time) and a delete button
const taskDetails = document.createElement('div');
taskDetails.classList.add('taskDetails');
// taskDetails.innerHTML = `<span>Target time:</span> ${taskTime} mins`;
taskDetails.innerHTML = `Target Time: ${taskTime} mins`;
const deleteButton = document.createElement('button');
deleteButton.innerText = 'Delete';
deleteButton.classList.add('deleteTask');

// Add task details and delete button to the task item
taskItem.appendChild(taskDetails);
taskItem.appendChild(deleteButton);

// Add the task item to the task list
taskList.appendChild(taskItem);

// Clear the input field and update the total tasks count
taskInput.value = '';
totalTasks++;
updateProgress();
} else {
// Display an alert if the input is invalid
alert('Please enter valid time');
}

}

// taskList(parent)
// -> taskItem(child of taskList and parent of below 4)
// ->checkbox(child of taskItem)
// ->label(child)
// ->taskDetails(child)
// ->deleteButton(child)

// Function to handle task actions (deleting using button or marking as
completed in checkbox)
function handleTaskActions(event) {
const target = event.target;

if (target.classList.contains('deleteTask')) {
// Remove the task if the delete button is clicked
```

```
target.parentElement.remove(); //taskItem is removed from taskList
totalTasks--;
updateProgress();
} else if (target.tagName.toLowerCase() === 'input' && target.type ===
'checkbox') { //tagname -> input,div,h1,p etc
target.parentElement.classList.toggle('completed'); // adds if
absent, removes if present

// If the task is completed, set its style to green and bold
if (target.parentElement.classList.contains('completed')) {
target.parentElement.style.color = 'green';
target.parentElement.style.fontWeight = 'bold';
} else {
// If the task is not completed, remove the green style color
target.parentElement.style.color = '';
target.parentElement.style.fontWeight = '';
}

// Count the number of completed tasks and update the progress
completedTasks = document.querySelectorAll('li.completed').length;
updateProgress();

// Prompt for actual time spent if the task is marked as completed
if (target.parentElement.classList.contains('completed')) {
const actualTime = parseFloat(prompt('Enter the time spent on task(in
minutes):'));
if (!isNaN(actualTime) && actualTime > 0) {
// this will create a section for actual time spent
const timeSpent = document.createElement('div'); //I will work on
this...to replace div element
timeSpent.innerHTML = `Time Spent: ${actualTime} mins`;
target.parentElement.querySelector('.taskDetails').appendChild(timeSpent);
}
else {
alert('Please enter valid time'); //if input is not valid
}
} else {
// Remove the actual time spent element if the task is incomplete
const timeSpentElement = target.parentElement.querySelector('.taskDetails
div');
```

```

if (timeSpentElement) { //if timeSpentElement=something
timeSpentElement.remove(); //remove it
}
}
}
}

// Function to update the progress and display it above
function updateProgress() {
// Count the total tasks and completed tasks at the current moment
const totalTasks = document.querySelectorAll('li').length;
const completedTasksNow =
document.querySelectorAll('li.completed').length;

const progressPercentage = totalTasks === 0 ? 0 :
Math.floor((completedTasksNow / totalTasks) * 100); //(smartly avoided the
division by 0 hehe)
//if (tasks ===0)? true condition : false condition;
// quickly Update the content of the progressValue element with the
calculated percentage
progressValue.textContent = `${progressPercentage}%`;
}

//END OF THE CODE

```

QUESTIONS :

Great, let's discuss your JavaScript code for the "Tasks Management Tool" project. I'll ask you questions to better understand how your code works.

1. What is the purpose of the variables `taskInput`, `taskList`, `addTaskButton`, and `progressValue`?
2. What do the variables `completedTasks` and `totalTasks` keep track of in your project?

3. You have added event listeners to `addTaskButton` and `taskList`. What actions trigger these event listeners, and what functions are called when these events occur?
4. Explain the `addTask` function. How does it add a new task to the task list, and what input validation does it perform?
5. Within the `addTask` function, you create a new list item (`taskItem`) and add a checkbox and label to it. What is the purpose of the checkbox and label?
6. The `addTask` function creates a section for task details. In your code, you initially set the `innerHTML` to `Target time: \${taskTime} mins`. Why did you comment this line and use `taskDetails.innerHTML = "Target Time: \${taskTime} mins";` instead?
7. Describe the purpose of the `handleTaskActions` function. What actions can be handled by this function?
8. When a task is marked as completed, what style changes are applied to the task item?
9. Explain how you handle user input for the actual time spent on a task in the `handleTaskActions` function.
10. In the `updateProgress` function, what does it calculate and update on the webpage?

Feel free to provide detailed answers to these questions to help me better understand your project's code.

ANSWERS:

1. The purpose of these variables is as follows:
 - `taskInput`: It references the HTML input element where users can enter task descriptions.
 - `taskList`: It references an HTML unordered list (``) where the tasks will be displayed.
 - `addTaskButton`: It references a button element used to add new tasks.
 - `progressValue`: It references an HTML element where the progress percentage is displayed.

2. `completedTasks` and `totalTasks` are used to keep track of the number of completed tasks and the total number of tasks, respectively. `completedTasks` counts the number of tasks marked as completed, and `totalTasks` counts all the tasks, whether completed or not.

3. Event listeners are added as follows:

- `addTaskButton`: The event listener is triggered when the "Add Task" button is clicked. It calls the `addTask` function to add a new task.
- `taskList`: This event listener handles various actions within the task list, such as deleting tasks and marking tasks as completed. It calls the `handleTaskActions` function.

4. The `addTask` function serves the purpose of adding a new task to the task list. Here's a breakdown of its functionality:

- It retrieves the task description and target time entered by the user.
- It validates the input to ensure that the task description is not empty and that the target time is a positive number.
- If the input is valid, it creates a new list item (`taskItem`) for the task and adds a checkbox and label for the task description.
- It also creates a section for task details, such as the target time, and adds a delete button.
- The task item is then added to the task list.
- The input field is cleared, and the total tasks count is updated.
- The progress is updated to reflect the change.

5. The checkbox and label are used for displaying the task description and allowing users to mark tasks as completed or not. The checkbox serves as a visual indicator, and the label displays the text of the task.

6. In your initial code, you used `taskDetails.innerHTML = Target time: ${taskTime} mins;`. This code creates a new `span` element for the "Target time" label and appends the `taskTime` variable within the same element. However, you commented this line and used `taskDetails.innerHTML =`

"Target Time: \${taskTime} mins"; instead, which simplifies the task details display. The new line doesn't use a `span` element but provides a more straightforward presentation of the task's target time.

7. The `handleTaskActions` function is responsible for handling two main actions:

- Deleting tasks: If the user clicks the "Delete" button for a task, it removes the task from the list and updates the total task count and progress.
- Marking tasks as completed or uncompleted: When a user clicks the checkbox, the function toggles the class 'completed' on the parent task item. It also applies or removes styling changes accordingly. If a task is marked as completed, the user is prompted to enter the actual time spent.

8. When a task is marked as completed, it receives the following style changes:

- Background color: It turns a light green color (#d3ffd3).
- Text color: It becomes green.
- Font weight: It is set to bold.

9. To handle user input for the actual time spent on a completed task, the code within the `handleTaskActions` function:

- Prompts the user to enter the actual time spent.
- Validates the input to ensure that it's a positive number.
- Creates a section for actual time spent and appends it to the task details if the input is valid.
- Displays an alert if the input is invalid.
- If the user unmarks a completed task, the actual time spent element is removed.

10. In the `updateProgress` function:

- It counts the total number of tasks and the number of completed tasks at the current moment.
- It calculates the progress percentage by dividing the number of completed tasks by the total tasks (avoiding division by zero).
- The progress percentage is then displayed in the `progressValue` element on the webpage.

I hope these detailed answers provide a clear understanding of your project's JavaScript code! If you have more questions or need further clarification, feel free to ask.

My Doubts:



taskList.addEventListener('click', handleTaskActions);
....when this line will work

< 1 / 2 >



The line `taskList.addEventListener('click', handleTaskActions);` sets up an event listener on the `taskList` element, which listens for click events. It will work when the `taskList` element or any of its child elements are clicked.

In this context, `taskList` is likely an HTML element that contains a list of tasks, possibly a `

` or `

` element. When any of the tasks within the list or the list itself is clicked, the `handleTaskActions` function will be called to handle the click event.

The purpose of this event listener is to manage actions related to tasks, such as marking a task as completed or deleting a task, as described in the `handleTaskActions` function. It allows the user to interact with the tasks by clicking on them or on the associated buttons (e.g., checkboxes and delete buttons) within the list.





const taskText = taskInput.value; ...what value will be here ... i mean how it works?

< 2/2 >



In the code `const taskText = taskInput.value;`, `taskInput` is a reference to an HTML input element, likely an input field where you can enter a task description. The `value` property of an HTML input element contains the current value entered by the user.

So, when this line of code is executed, `taskText` will hold the value that the user has entered into the `taskInput` element, which is the task description text that they have typed in the input field.

For example, if the user has entered "Buy groceries" in the input field, then `taskText` will be assigned the string "Buy groceries".

This value can then be used to create a new task with the description provided by the user.



```
input[type="checkbox"] {  
width: 20px;  
height:20px; /* size for check box...later added*/  
}
```

This added to increase size of checkbox

is label means that box of taskItem element ? bcz when i changed css styles size of label...the box size got increased



No, in your code, the "label" element doesn't refer to the box of the `taskItem` element. In the context of your code, the "label" element is used to display the text description of the task. It's not a box or container that contains the entire task item.

The `label.textContent = taskText;` line sets the text content of the label to the value of `taskText`, which is the description of the task entered by the user. This label is associated with a checkbox input, which is a common way to create clickable labels for checkboxes. When you click on the label, it toggles the associated checkbox.

If you changed the size of the label using CSS, and the box size of the `taskItem` increased, it's likely due to other CSS rules affecting the layout, such as the overall styling of the `taskItem` or its parent elements. The label's size change should not directly affect the box size of the `taskItem`. If you'd like to control the size of the entire `taskItem` element, you need to modify the CSS for that specific element or its container.



```
// Create a section for task details (target time) and a delete button
const taskDetails = document.createElement('div');
taskDetails.classList.add('taskDetails');
// taskDetails.innerHTML = `<span>Target time:</span> ${taskTime} mins`;
taskDetails.innerHTML = `Target Time: ${taskTime} mins`;
const deleteButton = document.createElement('button');
deleteButton.innerText = 'Delete';
deleteButton.classList.add('deleteTask');
```

for what this is written ?



This code snippet is part of a function called `addTask`, which is responsible for adding a new task to a task list on a web page. Let me break down the purpose of this code:



1. `const taskDetails = document.createElement('div');`: This line creates a new `div` element in the HTML document, which will be used to hold additional details related to the task being added.
2. `taskDetails.classList.add('taskDetails');`: This line adds the CSS class "taskDetails" to the `taskDetails` div element. The class "taskDetails" likely has associated CSS rules that define the appearance and styling of this element.
3. `taskDetails.innerHTML = 'Target Time: ${taskTime} mins';`: Here, the `innerHTML` property is used to set the content of the `taskDetails` div. It creates a text element displaying the target time for the task, which is taken from the `taskTime` variable, and adds " mins" to indicate that the time is in minutes. The text "Target Time: " is prepended to the `taskTime` value.
4. `const deleteButton = document.createElement('button');`: This line creates a new `button` element. This button will allow users to delete the task.
5. `deleteButton.innerText = 'Delete';`: The text displayed on the delete button is set to "Delete."
6. `deleteButton.classList.add('deleteTask');`: The CSS class "deleteTask" is added to the delete button. This class is likely used for styling and potentially for identifying delete buttons in the code.

In summary, this code creates a container (`taskDetails`) for additional task-related details (target time) and a "Delete" button for each task being added to the list. These elements are dynamically generated in the HTML document and provide a structured and visually appealing way to display task information and provide user interaction.



`classList` is a property of DOM elements in JavaScript that represents the classes of an HTML element as a collection (more specifically, a DOMTokenList). It provides methods and properties for working with the CSS classes of an element. You can use it to manipulate the classes of an HTML element dynamically.

Here are some common methods and properties provided by **`classList`**:

1. **`add(className)`**: Adds one or more class names to the element.
2. **`remove(className)`**: Removes one or more class names from the element.
3. **`toggle(className)`**: Toggles a class; adds it if it doesn't exist, removes it if it does.
4. **`contains(className)`**: Checks if a class name is present on the element.
5. **`item(index)`**: Returns the class name at a specific index.
6. **`length`**: Returns the number of classes associated with the element.

`classList` is particularly useful when you want to add or remove classes from an element based on user interactions or other events. This is commonly used in web development to control the styling and behavior of elements dynamically. It makes it easy to manipulate CSS classes without directly dealing with the string of class names in the element's **`className`** property.

⟳ Regenerar

1. **Inline Element:** `` is an inline HTML element, which means it doesn't create a new line and flows within the content like normal text. It doesn't add any line breaks before or after the content it encloses.
2. **Styling:** One of the primary uses of `` is for styling specific portions of text. You can apply CSS styles, such as colors, fonts, or text decorations, to the content enclosed within a `` element.

Example:

```
html Copy code  


<p>This is a <span style="color: red;">red</span> word.</p>


```

3. **Scripting:** In JavaScript, you can select and manipulate text within a `` element more easily than with regular text. It can be used as a convenient way to target specific content for interaction.

Example:

```
html Copy code  
and a message ▶ ↻
```

```
// Function to handle task actions (e.g., deleting or marking as completed)  
function handleTaskActions(event) {  
const target = event.target;
```

case, it's triggered when the user interacts with elements within the `taskList`.

2. It receives the `event` object as its parameter, allowing it to access information about the event, including the target element that triggered the event.
3. It first identifies the target element by accessing `event.target`. The `target` element is the one the user clicked on or interacted with.
4. It then checks if the `target` element belongs to one of two categories:
 - If it has a class of "deleteTask," it means the user clicked on the "Delete" button associated with a task, and the function proceeds to remove that task.
 - If the `target` element is an `` element (a checkbox), and its `type` is "checkbox," it means the user interacted with a task checkbox. The function proceeds to mark the task as completed and may prompt the user for additional information, such as the actual time spent on the task.
5. Depending on the type of interaction, the function either removes the task, marks it as ↓ completed, updates its appearance (e.g., making the text green and bold for completed



and what is .target which is attached to the event like event.target ??



In JavaScript, the `event` object contains a property called `target` which represents the DOM element that triggered the event. When an event occurs, such as a click event, the `target` property is set to the DOM element that was interacted with, like a button or a checkbox.

For example, if you have a click event on a button element, `event.target` would be a reference to that button element. This allows you to access and manipulate the properties and attributes of the element that triggered the event.

In your code:

javascript

Copy code

```
const target = event.target;
```

```
what if i use this "else if ( target.type === 'checkbox' ) {" instead of this line  
" else if (target.tagName.toLowerCase() === 'input' && target.type === 'checkbox') { "
```

The line `else if (target.type === 'checkbox')` would work if you are sure that the event target is always going to be a checkbox input element.

However, using `target.tagName.toLowerCase() === 'input' && target.type === 'checkbox'` is more specific. It ensures that the event target is both an input element and specifically a checkbox input. This way, your code won't accidentally handle clicks on other types of input elements if they are added to your document later.

So, while `target.type === 'checkbox'` might work in your current context, using `target.tagName.toLowerCase() === 'input' && target.type === 'checkbox'` is more precise and safer, especially if you plan to expand or modify your code in the future. It's a good practice to be as specific as possible to avoid unexpected behavior.

```
else if (target.tagName.toLowerCase() === 'input' && target.type === 'checkbox') {  
    // Toggle the class 'completed' for the task item  
    target.parentElement.classList.toggle('completed');
```

what is tagName ?

< 10 / 10 >



In the provided code, `target.tagName` is a property of the `target` element. It represents the HTML tag name of the element in lowercase. For example, if `target` refers to an HTML `` element, `target.tagName` would be the string "input" in lowercase.

Here's a breakdown of the code you mentioned:

1. `target.tagName.toLowerCase()`: This part accesses the `tagName` property of the `target` element, which tells you the tag name of the HTML element. For example, if the `target` element is an input element, it would be "INPUT".
2. `toLowerCase()`: This is a method that is called on the tag name, and it converts the tag name to lowercase. This is done to ensure that you're performing a case-insensitive comparison. In HTML, tag names are not case-sensitive, so "input" and "INPUT" should be treated as equivalent.

toggle means what? i am very confused about this word ?



In programming, "toggle" typically means to switch between two states or values. When you "toggle" something, you change it from one state to another and vice versa, and you can do this repeatedly.

In the specific context of the code you provided,

`'element.classList.toggle('completed')'`, it means:

1. If the element currently has the CSS class "completed," it will be removed (toggled off).
2. If the element does not have the "completed" class, it will be added (toggled on).

It's like turning a light switch on and off. When the switch is in the "on" position, you have the light, and when it's in the "off" position, the light is turned off. Toggling means changing the state, just like flipping the switch.



In the code, this toggling is used to mark tasks as completed or uncompleted when a

It's like turning a light switch on and off. When the switch is in the "on" position, you have the light, and when it's in the "off" position, the light is turned off. Toggling means changing the state, just like flipping the switch.

In the code, this toggling is used to mark tasks as completed or uncompleted when a checkbox is clicked. If the task has the "completed" class, it means it's marked as completed, and clicking the checkbox will remove the class (toggle off), marking it as uncompleted. Clicking the checkbox again will add the class (toggle on), marking it as completed.



can we say toggle creates class virtually

< 13 / 13 >



Yes, you can think of the `'toggle'` method as virtually creating a class. Here's how it works in the code you provided:

1. `'target.parentElement.classList.toggle('completed')'`: This line toggles the class `'completed'` on the parent element of the target. If the class `'completed'` is already present, it will be removed; if it's not present, it will be added. This effectively toggles the state of the task, marking it as completed or uncompleted.
2. If the class `'completed'` is added to the parent element, the task is considered completed, and the following styles are applied:
 - `'target.parentElement.style.color = 'green';'`: Sets the text color of the task to green.
 - `'target.parentElement.style.fontWeight = 'bold';'`: Sets the font weight of the task to bold.

3. If the class `'completed'` is removed from the parent element, the task is considered

Project Explanation

[HTML] → div, h1, input, button
, ul,

class = "Container"

Step 1 class?

<input> ⇒

id = taskInput
id = addTask

<input type = "text" id = "taskInput"
placeholder = "Enter a task" >

button

<button id = "addTask" > Add Task </button>

id = taskList

15 id = "progress"
& id = "progressValue"

<div id = "progress" > Progress: ~~0%~~

 0%

20 </div>

<body>

<div id = "progress" > Progress: 0% </div>

<div class = "container" >

25 <h1> Enhance Your Productivity </h1>

<input type = "text" id = "textInput" placeholder = "Enter..." >

<button id = "addTask" > Add Task </button>

<ul id = "taskList" >

30 </div>

</body>

</html>

[JavaScript] → DOM, event listeners, addEvent, handling, cloning, etc.

- (i) store elements Id's into ^(const) variables.
- (ii) add event listeners

addTaskButton.addEventListener('click', addTask);

function addTask() {

const taskText = taskInput.value;
const taskTime = .

if (input is valid) {

const taskItem = document.createElement('li') - ^{element created}
taskItem.innerHTML = taskText;

// ~~create~~ create section for details of task
→ target time

→ add element to the class

(taskDetails) (taskDetails)

→ taskDetails.classList.add('taskDetails');

→ create delete button

→ append (i) taskDetails & (ii) deleteButton
to the taskItem (each)

then append that taskItem to
the taskList

20/10/2023

Comlin Page
Date / /

3/10/2023
Hans

→ now clear the entered data, to take next entry

Also before clearing, increase no. of totalTasks & update the progress.

- Handling Task Actions

before this, I am going to learn 'classList' prop

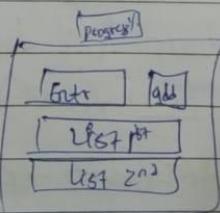
> const element = document.getElementById('myElement');
element.classList.add('active', 'highlight');

- remove (class1, class2, ...)
- toggle (class)
- contains (class)
- item (index)
- length

— o — x — InShot — x — o

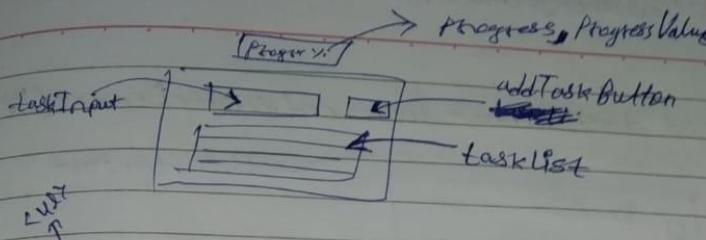
- Variables ⇒ taskInput, taskList, addTaskButton, progress

- functions ⇒ addTask(), handleTaskActions();
& updateProgress();



3/10/2023
Harm

Camlin Page
Date / /



taskList.addEventListener('click', handleTaskActions);

→ handleTaskActions() function will be called when we click taskList or any of its child.

addTaskButton.addEventListener('click', addTask);

→ addTask() function will be called when we click addTask button.

• Inside addTask button :-

const taskText = taskInput.value;

Value is HTML property
of input element

const taskTime = parseFloat(prompt(...));

→ parseFloat is used to convert String into number.
→ prompt is used to display dialog box

if(taskText.trim() != '' && !isNaN(taskTime)) {
 && taskTime > 0 } 3 conditions

→ trim() function removes blank spaces & checks if it's empty string or not

→ isNaN() = is not a number

!isNaN(taskTime) = taskTime is a number

जैसा कीना condition follow होता जाता है 'li' element create करे
else enter valid input

Yesterday created link for this proj.
⇒ <https://abhishekjachak21.github.io/TaskManagementTool/>

const taskItem = document.createElement('li');
taskItem.appendChild(label.textContent);
taskItem.appendChild(input);

- input type "text" ⇒ [hello]
- input type "checkbox" ⇒ [] [x]
- input type "password" ⇒ [.....]

```
const checkBox = document.createElement('input');
checkBox.type = 'checkbox';
const label = document.createElement('label');
label.textContent = taskText;
```

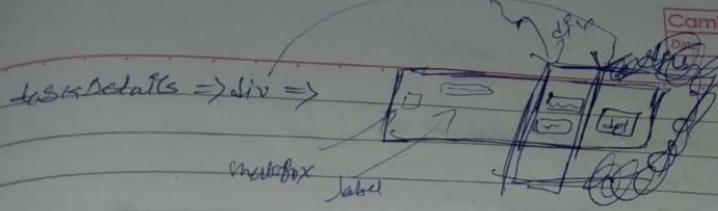
taskItem.appendChild(checkBox);
taskItem.appendChild(label);

taskItem.appendChild(label);
taskItem.appendChild(checkBox);

// Section for ~~task~~ Details (delete, target time, time spent)

```
const taskDetails = document.createElement('div');
taskDetails.classList.add('taskDetails');
taskDetails.innerHTML = `Target Time : ${taskTime} mins`;
const deleteButton = document.createElement('button');
deleteButton.innerText = 'Delete';
deleteButton.classList.add('deleteTask');
```

2023.10.26 00:11

taskDetails => div =>


 - classList is prop of DOM elements
 - ↳ it provides methods for working with CSS classes of elem
 ↳ add, remove, toggle, contains, item, length
 ↳ toggle = toggles a class
 = adds ↳ it IF it doesn't exist
 , removes ↳ if it does.

 - innerHTML & innerText are two diff prop
 ↳ NewItem
 & innerText
 ↳ 'NewItem'

 - const deleteButton = doc.createElement('button')
 taskItem.appendChild(taskDetails)
 ↳ (delButton) ↳ [del]

 ↳ Now • taskItem (li) [Parent]
 ↳ checkbox ← child
 ↳ label
 ↳ taskDetails
 ↳ deleteButton

 now, one item completed with all its attributes
 so tie it to the list (ul) bcz ↳
 ↳ taskList.appendChild(taskItem);

Camlin Page
Date / /

taskInput.value = '';
totalTasks++;
updateProgress();

do
clear
this TIP

→ ↑ the no. of total Tasks
→ helps to ↑ avg data of progress.

→ [Add] Button
← Done with the 1st AddTask function →

Inside listed items interactions [handleTaskActions() function]

function handleTaskActions(event) {
const target = event.target

→ If holds ~~the element~~ the element, user interacted with.

taskItem (Parent)
checkbox (child)
→ deletebutton

removes it from task list

target → Input (checkbox)
→ delete button

if (target.classList.contains('deleteTask'))
→ target.parentElement.remove();
totalTasks --;
updateProgress();

2023.10.26 00:12

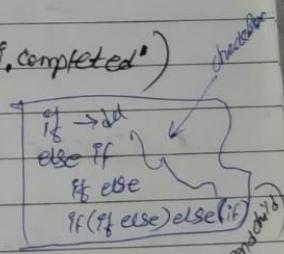
```
else if (target.tagName.toLowerCase() === 'input'
& target.type === 'checkbox') {
    target.parentElement.classList.toggle('completed');
```

→ tagName → input, h1, div etc
toLowerCase is to convert 'INPUT' to 'input'

→ target.parentElement.classList.toggle('completed')
 ↴
 ↴ input.toggle('completed')
 ↴ input.toggle('uncompleted')

→ toggle means add if absent
& remove if present

Completed Tasks = document.querySelectorAll('.if.completed')
 updateProgress();



→ it counts all li elements created.

→ acc. to this, progress updated.

if (!if completed) {

if (!isNaN(actualTime) & actualTime > 0) {

for (var i = 0; i < completed.length; i++)
 if (completed[i].innerHTML)

const timespent = document.createElement('div')

timespent.innerHTML = 'Time Spent: ' + [actualTime] + ' min'

target.parentElement.querySelector('.taskDetails').appendChild(timespent)

→ ~~taskItem~~ (taskItem).taskDetails (Parent)

→ timespent (child) now

```

if (complete)
  - ask actual time
  - if (actual time > 0)
    show it, add it to 'target time'
  else
    enter null
else (if incomplet) {

```

```

const timespentEle = target.parentElement.querySelector
if (timespentEle) {  
  timespentEle.remove();  
}
}

```

- remember? the child of taskdetails class
 taskdetails (div) (parent)
 ↳ timespent (div) (child)

→ So even if, timespentEle = something
 remove it from completed class

- Inside updateProgress() function

$$\%age = \frac{\text{li}.completed}{\text{li}} \times 100$$

const progressPercentage = (value) ? true : false

const progressValue = (totalTasks == 0) ? 0 : Math.floor(

:Math.floor((completedTasks/totalTasks)*100));

progressValue.textContent = `\${progressValue}%`

- अंतम् (5:05 PM)