### **The "Name" field:**

When the page loads, the first form field should be in focus, meaning it should have a flashing cursor and/or be highlighted in some way.

1. Create a variable to reference the “Name” <input type="text"> element and log the variable out to the console to ensure the correct element is being referenced.
2. Use the variable and dot notation to set the focus property of the element to true.

Now save and refresh the page, and the “Name” field should have the focus state.

### **4. "Job Role" section:**

Adjacent to the "Job Role" menu on the page is the "Other job role" field. This field should be hidden by default, and only be displayed if the user selects the "Other" option from the "Job Role" drop-down menu. And if the user selects any other option, the "Other job role" field should be hidden from view.

Programming the "Job Role" section requires working with the following elements:

* The "Job Role" <select> element
* The "Other job role" <input type="text"> element

1. Create variables to reference the "Job Role" <select> element and the "Other job role" <input type="text"> element. Log the variables out to the console to ensure the correct elements are being referenced.
2. Use the "Other job role" variable to hide this element by default.
3. Use the variable for the "Job Role" menu to listen for the change event on this element. When a "change" is detected, use a conditional statement to check the value property of the element. The event.target statement will be helpful here. Log out the condition and the event.target’s value to inspect them. You’ll have to make a selection in the "Job Role" menu to print those log statements to the console.
4. In the conditional, if the value of the event.target is equal to "other", display the “Other job role” field. And if the value is anything, hide it.

Now save and refresh the page, and when the "Other" option is selected/deselected from "Job Role" menu, the "Other job role" field should be visible/hidden on the page.

### **5. "T-Shirt Info" section:**

The "T-Shirt Info" section has conditional options. The "Design" or "Theme" menu has two options, and the "Color" menu has six. But only three of the colors are available for each of the "Design" options. To prevent users from selecting an invalid color for a particular theme, the "Color" menu should be disabled by default. Once a theme is selected, the "Color" menu should be enabled, and the color options need to be displayed/hidden based on which theme the user has selected.

For some experience with the techniques you’ll use in this section, complete this short project Warm Up, [Selects and Options](https://teamtreehouse.com/library/fsjs-project-warm-up-selects-and-options).

Programming the "T-Shirt Info" section requires working with the following elements:

* The "Design" <select> element
* The "Color" <select> element
* The option element children of the the "Color" <select> element

1. Create variables to reference the "Design" <select> element and the "Color" <select> element. The color option elements can be referenced by targeting the .children property of the "Color" <select> element. Log the variables out to the console to ensure the correct elements are being referenced.
2. Use the color variable and dot notation to set the disabled property of the element to true.
3. Use the variable for the "Design" or "Theme" menu to listen for the change event on this element.
4. In the event listener, enable the “Color” <select> element that was previously disabled.
5. Also in the event listener, loop over the option element children of the "Color" <select> element. The children property will be helpful here.
6. In the loop, create variables to reference the following two items:
   * The value of the event.target
   * The "data-theme" attribute of the loop’s current option element. The getAttribute method will be helpful here.
7. Log out the two variables that were just created to inspect them. You’ll have to make a selection in the "Design" menu to print those log statements to the console.
8. Still in the loop, create a conditional that checks if the two variables that were just created are equal to one another. If they are, set the hidden property of the loop’s current option element to false, and set the selected attribute of the loop’s current option element to true. And if the two variables are not equal to one another, set the hidden property of the loop’s current option element to true, and set the selected attribute of the loop’s current option element to false.

**Note:**

* You can use dot notation to update the hidden property. But you’ll need to use the setAttribute or removeAttribute methods to update the selected attribute.
* Using dot notation or the getAttribute method, the hidden property and the selected attribute of an element can be logged out to the console for inspection.

Now save and refresh the page, and when a theme is selected, the "Color" field should be enabled and it’s value should update along with the options in the "Color" drop-down menu.

### **6. "Register for Activities" section:**

The total cost of the selected activities in the "Register for Activities" section should be totaled and displayed for the user.

For some experience with the techniques you’ll use in this section, complete this short project Warm Up, [Checkboxes](https://teamtreehouse.com/library/fsjs-project-warm-up-checkboxes).

Programming the total cost of the "Register for Activities" section requires working with the following elements:

* The "Register for Activities" <fieldset> element
* The "Total: $" <p> element

1. Create variables to reference the "Register for Activities" <fieldset> element and the "Total: $" <p> element. Log the variables out to the console to ensure the correct elements are being referenced.
2. Create another variable to store the total cost of the activities and give it an initial value of 0.
3. Use the variable for the "Register for Activities" section to listen for the change event on this element.
4. Inside the event listener, create a variable to store a reference to the "data-cost" attribute of the event.target. The getAttribute method will be helpful here. This will provide the cost of the activity that was just clicked. But this value needs to be a number rather than a string, so you’ll need to perform a type conversion on this value before working with it. The unary plus operator, +, will be helpful here. To test this value and its type, log out the variable, and log it out a second time with the typeof operator to ensure it is a number and the value you expect it to be. You’ll have to make a selection in the "Register for Activities" section to print those log statements to the console.
5. Still inside the event listener, create a conditional to determine if the event.target was just checked or unchecked. The checked property will be helpful here. If the event.target was checked, then add the "data-cost" of the event.target to the total variable that was created earlier. If the event.target was unchecked, then subtract the “data-cost”. To test this, you can log out the total cost variable as well as the checked property of the event.target. You’ll have to select activities to see this log statement as well.
6. Lastly, update the innerHTML of the “Total:” <p> element with the new total cost, but mind the formatting so that the end result still resembles the initial text: "Total: $0".

Now save and refresh the page, and when an activity is checked/unchecked, the total cost text below the activity section should update in real time.

### **7. "Payment Info" section:**

The preferred or most common payment method option should be selected and the corresponding payment form sections should be displayed by default, while the other payment form sections should be hidden.

Programming the "Payment Info" section requires working with the following elements:

* The "I'm going to pay with:" <select> element
* The <div> element with the id of "credit-card"
* The <div> element with the id of "paypal"
* The <div> element with the id of "bitcoin"

1. Create variables to reference the above elements, and log them out to the console to confirm their identity.
2. Use the "paypal" and "bitcoin" variables above to hide these elements initially.
3. Use the payment variable above to target the element’s second child element and give it the selected property. The .children property and the setAttribute method will be helpful here.
4. Use the payment variable above to listen for the change event on this element. When a change is detected, display the <div> element with the id that matches the value of the event.target element. And hide the other two <div> elements.

Now save and refresh the page, and when the payment method option is updated in the drop-down menu, the payment sections in the form will update accordingly.

### **8. Form Validation:**

Form submission should be prevented if one or more of the required fields or sections is not filled out correctly. This requires programming the form to listen for the submission event, and then to test the value or condition of the required form fields or sections when submission is detected, and finally allow/prevent submission based on the results of those tests.

**Note:** Form submission behavior will differ depending on whether you’re running the project with a local server, or just viewing the files in the browser. It is recommended that you [view the files in the browser](https://teamtreehouse.com/library/%E2%80%9Dhttps:/teamtreehouse.com/library/preview-files-in-a-browser%E2%80%9D) instead of serving them locally. This helps facilitate development and testing, and this is how the project will be reviewed.

For some experience with the techniques you’ll use in this section, complete this short project Warm Up, [Form Input Validation](https://teamtreehouse.com/library/fsjs-project-warm-up-form-input-validation-2).

Programming the "Form Validation" section requires working with the following elements:

* The "Name" <input type="text"> element (should already have a variable)
* The "Email Address" <input type="text"> element
* The "Register for Activities" <fieldset> element (should already have a variable)
* The "Card number" <input type="text"> element
* The "Zip code" <input type="text"> element
* The "CVV" <input type="text"> element
* The <form> element

1. Create variables for the above elements that haven’t already had variables created for them, and log them out to ensure they are what you expect them to be.
2. Use the "form" variable to listen for the submit event.
3. Inside the event listener, use the name variable, dot notation and the value property to create a new variable that references the value of the “Name” field.
4. Create another variable to store the results of testing the name value variable that was just created. Regex will be helpful, and the regex course from this unit will demonstrate how to do this. This variable will equal true if the test passes, and false otherwise.
5. Log these variables out to the console to check the "Name" field’s value and if it passes the custom validation test. To see this log statement printed to the console, you’ll have to do three things:
   * Enter something into the “Name” field
   * Add a temporary event.preventDefault() statement to prevent the form from refreshing when the submit button is clicked.
   * Click the submit button
6. Still in the event listener, use the name validation test variable and an if statement to check if the name value is valid. If false, prevent the form from submitting with an event.preventDefault() statement. Otherwise, do nothing and allow the form to submit.

Repeat steps 3 through 6 above for each of the required fields. Review the project instructions for the list of required fields or sections and the validation requirements of each. And remember that the credit card fields should only be validated if "credit card" is the selected payment method.

**Note:** Inside the event listener is ultimately where the required fields or sections will be tested. But rather than creating all the tests there, it can be helpful to create helper functions for each required field that can then be called in the event listener to do the testing and return true or false depending on whether the field is valid or not.

If you added an extra `event.preventDefault()` statement for testing purposes, it should be removed. Here, the form should only be prevented from submitting if one or more of the required form fields is invalid.

Now save and refresh the page, and when the submit button is clicked, the form should only submit and refresh if all the required fields are valid.

### **9. Accessibility:**

Web forms typically notify users of certain events or circumstances, like when an element is in focus, or if a field is invalid. When creating these indications it is important that they can be perceived by all users, even those with impairments.

For some experience with the techniques you’ll use in this section, complete this short project Warm Up, [Form Input Validaiton Error Indications](https://teamtreehouse.com/library/fsjs-project-warm-up-input-validation-error-indications).

When the checkboxes in the "Register for Activities" section are in focus, there’s little to no indication. So to improve accessibility, the checkboxes’ parent label elements should receive additional styles when their respective checkboxes are in focus.

1. Create a variable to reference the activities’ <input type=”checkbox”> elements, and log out the variable to ensure it is what you think it is.
2. Use the variable that was just created to loop over the activities’ checkboxes.
3. Inside the loop, program each activity to listen for the focus event and the blur event. These are two separate events and will need to be added and defined separately, but within the same loop.
4. When an activity checkbox triggers the focus event, that checkboxes’ parent element should have the "focus" className added to it. The classList property will be helpful here.
5. When an activity checkbox triggers the blur event, that checkboxes’ parent element should have the "focus" className removed from it. The classList property will be helpful here.

When an invalid form field or section prevents the form from submitting, there’s little to no indication. So to improve accessibility, add form input validation error indications that can be perceived by all users.

1. When validating a required form field, like the "Name" filed, and checking whether the field is valid or not, you’ll need to perform three tasks;
   * If the form field element is invalid:
   * Add the "not-valid" className to the element’s parent element.
   * Remove the "valid" className from the element’s parent element.
   * Display the last child of the element’s parent element. The lastElementChild property will be helpful here.
   * If the form field element is valid:
   * Add the "valid" className to the element’s parent element.
   * Remove the "not-valid" className from the element’s parent element.
   * Hide the last child of the element’s parent element. The lastElementChild property will be helpful here.

Now save and refresh the page, and when the submit button is clicked without all the required fields being correctly filled out, the user should be notified with clear form input validation error indications.