Hogwarts School of Codecraft and Spellscript

Introduction: Welcome, First-Year!

Welcome to the Hogwarts School of Codecraft and Spellscript! I am Professor Filius Flitwick, Head of the Charms Department. You, my young student, are about to learn the most powerful and flexible magic of all: **SpellScript** (known to Muggles as JavaScript!).

In this world, every web page is a piece of living parchment. With SpellScript, you can write the incantations to change it, make it interactive, and bring it to life.

But be warned! A dark magic is afoot. The malevolent **Lord Voldenull** and his **Syntax-Serpents** (nasty "Bugs") are trying to corrupt the digital world with broken, chaotic code.

Only you can master the logical charms and defensive arts needed to become a true Code-Weaver. Let's begin your first lesson!

Scroll I: The First-Year's Kit (JS Basics)

This scroll covers the absolute essentials: how to store magical information and perform simple transfigurations.

Quest 1: The Memory Vial

The Story (Docs - Left Side):

"Just like Professor Dumbledore stores his memories in a Pensieve, we need 'vials' to store our pieces of SpellScript," says Professor Flitwick. "We call these **Variables**."

"We have three types of vials:"

1. let: The standard, modern Memory Vial. You can change what's inside it.
   let studentName = "Harry Potter";
   studentName = "Hermione Granger"; // This is allowed!
2. const: A Sealed Vial. The value is "constant" and cannot be changed. This is perfect for magic you know will never change, like your own name or the name of our school.
   const schoolName = "Hogwarts";
3. **var**: An "Old, Leaky Vial." This is an ancient, clumsy way to store memories. It can cause... *unexpected* side effects. We won't be using it, as modern wizards always prefer let and const.

Your Quest (IDE - Right Side):

Professor Snape needs you to organize his potions cabinet labels.

1. **Declare a let variable** called potionStock and assign it the *number* 10.
2. Snape uses one potion. **Re-assign** potionStock to be 9.
3. **Declare a const variable** called potionMaster and assign it the *string* "Severus Snape".
4. To see your spells, you must conjure them to the **Great Hall Monitor** (the Console). Use the console.log() spell to show both your variables.

Example Code (in IDE):

```
// This is a spell-comment. The Script ignores it!
console.log("Organizing potions...");

// Your code goes here:
let potionStock = 10;
potionStock = 9;
const potionMaster = "Severus Snape";

// Log them to the monitor:
console.log(potionStock);
console.log(potionMaster);
```

Quest 2: Potion Ingredients (Data Types)

The Story (Docs - Left Side):

"Very good! But you can't just throw *anything* into a cauldron," Flitwick chirps. "Every piece of data has a *type*. Using the wrong one can turn your potion into a... well, a toad."

These are the most common **Data Types**:

1. **String:** Text, always in quotes. (e.g., "Wingardium Leviosa", "Gryffindor")
2. **Number:** Any number. (e.g., 10, 3.14159, -50)
3. **Boolean:** A "Truth Charm." It can only be true or false. (e.g., isEvil = true;, isBrave = false;)
4. **Undefined:** A vial that's been created but has nothing in it. It's an empty vial you forgot to fill.
5. **Null:** A vial you *intentionally* emptied.

You can check any vial's type with the typeof spell:

```
console.log(typeof potionStock); // This would log "number"
console.log(typeof potionMaster); // This would log "string"
```

Your Quest (IDE - Right Side):

You're brewing a Swelling Solution. You must verify the *type* of each ingredient before adding it.

1. A jar of bat spleens has the label 15. Create a variable spleenType and assign it the *type* of the value 15.
2. A bottle of puffer-fish eyes is marked with the *text* "Caution". Create a variable labelType and assign it the *type* of the value "Caution".
3. A note asks, "Is potion ready?" The answer is false. Create a variable statusType and assign it the *type* of the value false.
4. console.log() all three of your new "type" variables.

Example Code (in IDE):

```
let spleenValue = 15;
let labelValue = "Caution";
let statusValue = false;

// Your code goes here:
let spleenType = typeof spleenValue;
let labelType = typeof labelValue;
let statusType = typeof statusValue;

// Show the results:
console.log("Spleen type is: " + spleenType);
console.log("Label type is: " + labelType);
console.log("Status type is: " + statusType);
```

Quest 3: The Gringotts Vault (Strings & Template Literals)

The Story (Docs - Left Side):

"Often, we need to combine strings. The old way was with the + charm," says Flitwick.

```
let firstName = "Ron";
let lastName = "Weasley";
let fullName = firstName + " " + lastName; // "Ron Weasley"
```

"But this is clumsy! Modern wizards use **Template Literals**. They use back-ticks (`) instead of quotes and let you magically insert variables right into the string with ${...}!"

```
`let fullName = `${firstName} ${lastName}`;` // "Ron Weasley"
`let message = `Hello, ${studentName}! You have ${potionStock} potions left.`;`
```

This is *much* cleaner and is the preferred magic.

Your Quest (IDE - Right Side):

A Gringotts goblin needs you to help format a vault statement.

1. Create a const variable wizardName set to your name.
2. Create a let variable galleons set to 50.
3. Create a let variable sickles set to 25.
4. Using **Template Literals**, create a single string variable vaultStatement that reads: "Vault Statement for [Your Name]: You have 50 Galleons and 25 Sickles."
5. console.log() the vaultStatement.

Example Code (in IDE):

```
// Your code:
const wizardName = "Neville Longbottom"; // Change this to your name!
let galleons = 50;
let sickles = 25;

let vaultStatement = `Vault Statement for ${wizardName}: You have ${galleons} Galleons and ${sickles} Sickles.`;

console.log(vaultStatement);
```

Quest 4: The Restricted Section (Comparison & Logic)

The Story (Docs - Left Side):

"To enter the Restricted Section, you must meet certain *conditions*," Flitwick whispers. "SpellScript uses **Operators** to ask questions. They always result in a true or false Boolean."

- > (Greater than): 10 > 5 is true
- < (Less than): 10 < 5 is false
- >= (Greater than or equal to)
- <= (Less than or equal to)

"The Most Important Charm: === vs =="
"Pay attention! This is crucial!"

- == (Loose Equality): This old charm is... *unreliable*. It tries to "transfigure" types. 7 == "7" is true. This is bad!
- === (Strict Equality): This modern charm is precise. It checks *both* the value AND the type. 7 === "7" is false. **Always use the === charm!**
- !== (Strictly Not Equal): The opposite of ===.

Logical Operators (Combining Charms):

- && (AND): Both conditions must be true.
  isAwake && isReady
- || (OR): At least one condition must be true.
  hasBroom || hasApparated
- ! (NOT): Flips the value.
  !isEvil // If isEvil is false, this becomes true

Your Quest (IDE - Right Side):

A magical barrier blocks the Restricted Section. It will only open if you answer its riddles true.

1. Create riddleOne. Is the number 100 *strictly equal* to the string "100"?
2. Create riddleTwo. Is 50 *greater than or equal to* 49?
3. Create riddleThree. Is 10 *not strictly equal* to 10?
4. **Final Riddle:** Create canEnter. The barrier will open if riddleTwo is true **AND** riddleThree is false.
5. console.log() the value of canEnter.

Example Code (in IDE):

```
// Your code:
let riddleOne = 100 === "100"; // false
let riddleTwo = 50 >= 49;      // true
let riddleThree = 10 !== 10;   // false

let canEnter = riddleTwo && !riddleThree; // (true AND !(false)) -> (true AND true) -> true

console.log("Riddle One: " + riddleOne);
console.log("Riddle Two: " + riddleTwo);
console.log("Riddle Three: " + riddleThree);

console.log("Can I enter? " + canEnter);
```

Scroll II: Charms & Incantations (Functions & Logic)

You've learned the nouns. Now you'll learn the *verbs*—how to write reusable spells and make choices.

Quest 1: The Standard Book of Spells (Functions)

The Story (Docs - Left Side):

"You can't be expected to re-write a complex spell every time! We write them down in our Spellbook as **Functions**."

A **Function** is a named, reusable block of code. You "declare" the spell once, then "call" it (use it) whenever you want.

```
// 1. Declare the spell
function castLumos() {
  console.log("A bright light appears at your wand tip!");
}

// 2. Call the spell
castLumos(); // "A bright light appears..."
castLumos(); // "A bright light appears..."
```

Some spells need an *ingredient* or *target*. We call these **Parameters**:

```
function greetWizard(name) {
  console.log(`Well met, ${name}!`);
}

greetWizard("Malfoy"); // "Well met, Malfoy!"
greetWizard("Luna"); // "Well met, Luna!"
```

The return Charm:
"What about a spell that gives you something back?" Flitwick asks. "An Accio charm, for instance. We use the return keyword."

```
function addPoints(pointsScored) {
  let housePoints = 150;
  let newTotal = housePoints + pointsScored;
  return newTotal; // This sends the value OUT
}

let gryffindorPoints = addPoints(10); // 10 is the "argument"
console.log(gryffindorPoints); // This will log 160!
```

Once return is hit, the function stops.

Your Quest (IDE - Right Side):

1. **Declare a function** named castAlohomora. Inside, it should console.log("The lock clicks open!").
2. **Call** your new castAlohomora function.
3. **Declare another function** called calculateGalleons. It should take one *parameter* called sickles.
4. Inside calculateGalleons, create a variable totalGalleons that is the sickles divided by 17.
5. **return** the totalGalleons.
6. Create a variable myGalleons and set it to the result of *calling* calculateGalleons with an *argument* of 340.
7. console.log() your myGalleons.

Example Code (in IDE):

```
// Your first function:
function castAlohomora() {
  console.log("The lock clicks open!");
}

castAlohomora();

// Your second function:
function calculateGalleons(sickles) {
  let totalGalleons = sickles / 17;
  return totalGalleons;
}

let myGalleons = calculateGalleons(340);
console.log(`I have ${myGalleons} galleons.`); // Should be 20
```

Quest 2: The Sorting Hat's Logic (If / Else If / Else)

The Story (Docs - Left Side):

"A spell must often make a *choice*. The Sorting Hat is the perfect example of this logic. We use if, else if, and else."

```
let trait = "brave";

if (trait === "brave") {
  console.log("GRYFFINDOR!");
} else if (trait === "cunning") {
  console.log("SLYTHERIN!");
} else if (trait === "wise") {
  console.log("RAVENCLAW!");
} else if (trait === "loyal") {
  console.log("HUFFLEPUFF!");
} else {
  console.log("Hmm, a difficult choice...");
}
```

The spell checks each condition in order, and *only* runs the *first* one that is true. The final else is a fallback if nothing else matches.

Your Quest (IDE - Right Side):

Write a sorting charm for a new student.

1. Create a let variable studentTrait and set it to "wise".
2. Write an if...else if...else block just like the example above to sort the student.
3. console.log() the result.
4. **Try it again!** Change studentTrait to "loyal" and see the magic work.

Example Code (in IDE):

```
let studentTrait = "wise";

if (studentTrait === "brave") {
  console.log("GRYFFINDOR!");
} else if (studentTrait === "cunning") {
  console.log("SLYTHERIN!");
} else if (studentTrait === "wise") {
  console.log("RAVENCLAW!");
} else if (studentTrait === "loyal") {
  console.log("HUFFLEPUFF!");
} else {
  console.log("Hmm, a difficult choice...");
}
```

Quest 3: The Quick-Quill (Arrow Functions & Scope)

The Story (Docs - Left Side):

"The function keyword is standard, but modern wizards often use a shorthand: **Arrow Functions**! They are clean, fast, and very popular."

Standard Function:

```
function add(a, b) {
return a + b;
}
```

Arrow Function:

```
const add = (a, b) => {
return a + b;
};
```

"If the function is only one line, you can make it even shorter!"

```
const add = (a, b) => a + b; // Wow!
```

A Note on Scope:

"One last thing! Vials (let/const) created inside a function are local to that function. They disappear when the spell is finished. This is 'The Room of Requirement' rule—what happens in the room, stays in the room!"

```
function testScope() {
  let localSecret = "I am a secret";
  console.log(localSecret); // This works
}
// console.log(localSecret); // This will CRASH! The vial doesn't exist out here.
```

This is a *good* thing! It keeps your spells from interfering with each other.

Your Quest (IDE - Right Side):

1. Rewrite your castAlohomora spell from Quest 1 as an **arrow function** assigned to a const of the same name.
2. Rewrite your calculateGalleons spell as a **one-line arrow function**.
3. Call both new functions to prove they work!

Example Code (in IDE):

```
// Your code:
const castAlohomora = () => {
  console.log("The lock clicks open!");
};

const calculateGalleons = (sickles) => sickles / 17;

// Test them:
castAlohomora();
let myNewGalleons = calculateGalleons(51);
console.log(`I have ${myNewGalleons} galleons.`); // Should be 3
```

Scroll III: Magical Containers (Arrays, Loops, & Objects)

Your magic is growing! But you can't just leave your ingredients all over the floor. You need containers.

Quest 1: The Potions Rack (Arrays)

The Story (Docs - Left Side):

"To hold a *list* of items in order, we use a **Potions Rack**, which Muggles call an **Array**. We use square brackets []."

```
let ingredients = ["Wolfsbane", "Bezoar", "Unicorn Hair", "Dragon Liver"];
```

"To get an item, we use its number, called an index. WARNING! Indexes always start at 0!"

```
console.log(ingredients[0]); // Logs "Wolfsbane"
console.log(ingredients[2]); // Logs "Unicorn Hair"
```

"An Array has its own built-in magic:"

- ingredients.length: Tells you how many items are on the rack (e.g., 4).
- ingredients.push("Bat Spleens"): Adds an item to the *end* of the rack.
- ingredients.pop(): Removes the *last* item from the rack.

Your Quest (IDE - Right Side):

You need to pack your school trunk.

1. Create a new, *empty* **Array** called trunk.
2. Use .push() to add the following items: "Robes", "Wand", "Cauldron".
3. You packed too many books. Use .pop() to remove the last item. (Wait... that's the Cauldron! Oh well!)
4. You forgot your owl! Use .push() to add "Hedwig".
5. console.log() the final trunk Array.
6. console.log() the trunk.length.

Example Code (in IDE):

```
// Your code:
let trunk = [];

trunk.push("Robes");
trunk.push("Wand");
trunk.push("Cauldron");

trunk.pop(); // Removes "Cauldron"

trunk.push("Hedwig");

console.log("My trunk contains:");
console.log(trunk); // ["Robes", "Wand", "Hedwig"]
console.log(`I have ${trunk.length} items.`); // 3
```

Quest 2: The Great Hall's Candles (Loops)

The Story (Docs - Left Side):

"You can't possibly cast Lumos 100 times to light all the candles in the Great Hall. You need a spell to *repeat* magic! We call this a **Loop**."

The for Loop:
"This is the most common loop. It's perfect when you know exactly how many times you want to repeat a spell."

```
for (let i = 0; i < 10; i++) { ... }
```

It has three parts:

1. let i = 0: The Counter. We start at 0.
2. i < 10: The Condition. Keep looping *as long as* i is less than 10.
3. i++: The Increment. After each loop, add 1 to i.

This loop will run 10 times (for i=0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

"We can use this to loop over our Arrays!"

```
for (let i = 0; i < ingredients.length; i++) {
  console.log(`Checking ingredient: ${ingredients[i]}`);
}
```

Your Quest (IDE - Right Side):

1. Write a for loop that counts from 1 to 5 (Hint: start i = 1 and use <= 5) and logs "Casting spell number [i]!" for each.
2. Use your trunk Array from the last quest.
3. Write a for loop that loops through your trunk and console.log()s each item.

Example Code (in IDE):

```
// Quest 1:
console.log("Casting spells...");
for (let i = 1; i <= 5; i++) {
  console.log(`Casting spell number ${i}!`);
}

// Quest 2:
let trunk = ["Robes", "Wand", "Hedwig"];
console.log("Checking my trunk...");
for (let i = 0; i < trunk.length; i++) {
  console.log(`Item ${i+1}: ${trunk[i]}`);
}
```

Quest 3: The Monster Book of Monsters (Objects)

The Story (Docs - Left Side):

"An Array is a *list*, but what about a single, complex thing? Like a magical creature? For that, we use an **Object**. An object is a container of **key-value pairs**. We use curly braces {}."

```
let hippogriff = {
  name: "Buckbeak",
  health: 100,
  isTame: false,
  weakness: "Pride"
};
```

"To get a value, we use dot notation:"

```
console.log(hippogriff.name); // Logs "Buckbeak"
```

"You can also use bracket notation:"

```
console.log(hippogriff["health"]); // Logs 100
```

"You can even add **spells (Methods)** inside your objects!"

```
let hippogriff = {
  name: "Buckbeak",
  greet: function() {
    console.log("Buckbeak bows respectfully.");
  }
};
```

```
// Now we can call the method!
hippogriff.greet(); // "Buckbeak bows respectfully."
```

Your Quest (IDE - Right Side):

1. Create an **Object** called niffler.
2. Give it the following **properties (keys) and values**:
   - name: "Gildy"
   - color: "black"
   - pouchFull: false
   - shiniesFound: 0
3. console.log() the niffler's name.
4. The niffler finds a coin! **Update** the shiniesFound property to 1 and pouchFull to true.
5. Add a **method** called chatter. This function should console.log("Squeak! *clink*").
6. **Call** the niffler.chatter() method.

Example Code (in IDE):

```
// Your code:
let niffler = {
  name: "Gildy",
  color: "black",
  pouchFull: false,
  shiniesFound: 0,
  chatter: function() {
    console.log("Squeak! *clink*");
  }
};

console.log(`My niffler's name is ${niffler.name}`);

niffler.shiniesFound = 1;
niffler.pouchFull = true;

console.log(`Pouch is full? ${niffler.pouchFull}`);

niffler.chatter();
```

Scroll IV: The Marauder's Map (The DOM)

You've learned SpellScript. Now, you will learn to change the "Living Parchment" (the web page) itself. This is the magic of the **DOM (Document Object Model)**.

Quest 1: "I Solemnly Swear..." (Selecting Elements)

The Story (Docs - Left Side):

"This web page is your Marauder's Map. It's full of hidden elements. Before you can change them, you must *find* them."

"The most powerful finding spell is document.querySelector()."

- document.querySelector("#some-id"): Finds *one* element with an id. (The # is for ID).
- document.querySelector(".some-class"): Finds *one* element with a class. (The . is for class).
- document.querySelectorAll(".student"): Finds *all* elements with the class "student" and returns them in an Array-like list!

This spell returns the *element object*, which we store in a vial.

Your Quest (IDE - Right Side):

(Your IDE has a hidden, connected web page for this.)
On the page, there is a title with an id of "great-hall-title" and three p tags, all with a class of "house-banner".

1. Find the title element using querySelector with its ID. Store it in a variable called titleElement.
2. Find *all* the house banners using querySelectorAll with their class. Store them in a variable called banners.
3. console.log() the titleElement.
4. console.log() the banners list.

Example Code (in IDE):

```
// Your code:
let titleElement = document.querySelector("#great-hall-title");
let banners = document.querySelectorAll(".house-banner");

console.log("I found the title:");
console.log(titleElement);
console.log("I found the banners:");
console.log(banners);
```

Quest 2: "Mischief Managed" (Changing Elements)

The Story (Docs - Left Side):

"You *found* the elements! Now, let's *change* them."

- To change the text:
  titleElement.textContent = "The Yule Ball";
- To change its style (CSS):
  titleElement.style.color = "gold";
  titleElement.style.fontSize = "40px";
- To change an attribute (like a portrait's image):
  let portrait = document.querySelector("#fat-lady");
  portrait.src = "fat-lady-singing.png";

Your Quest (IDE - Right Side):

The Great Hall title is boring. Let's fix it.

1. **Find** the element with the ID "great-hall-title" and store it.
2. **Change its textContent** to "Welcome, First-Years!".
3. **Change its style.color** to "darkred" (for Gryffindor, of course).
4. **Change its style.backgroundColor** to "gold".

Example Code (in IDE):

```
let titleElement = document.querySelector("#great-hall-title");

// Your code:
titleElement.textContent = "Welcome, First-Years!";
titleElement.style.color = "darkred";
titleElement.style.backgroundColor = "gold";
```

Quest 3: The Revealing Charm (Events)

The Story (Docs - Left Side):

"This is the real magic! We can make the map *react* to our touch. We set a 'magic trap'—an **Event Listener**—on an element. When a user *clicks* it, our spell fires!"

The spell looks like this:

```
elementToWatch.addEventListener("eventName", functionToRun);
// 1. The spell to run
function revealFootsteps() {
  console.log("Mischief Managed!");
}
```

```
// 2. Find the element (the map)
let mapElement = document.querySelector("#map");
// 3. Set the trap!
mapElement.addEventListener("click", revealFootsteps);
```

Now, every time the map is clicked, "Mischief Managed!" will appear in the console.

Your Quest (IDE - Right Side):

There is a hidden button on the page with an ID of "spell-button".

1. **Find** the button element and store it in a variable.
2. **Declare a function** named onCastSpell.
3. Inside onCastSpell, **console.log()** the message "Expecto Patronum!".
4. **Add an event listener** to your button. Listen for the "click" event, and tell it to run your onCastSpell function.
5. After your code runs, **click the button** and watch the console!

Example Code (in IDE):

```
// Your code:
let myButton = document.querySelector("#spell-button");

function onCastSpell() {
  console.log("Expecto Patronum!");
}

myButton.addEventListener("click", onCastSpell);

console.log("Event listener is set! Click the button.");
```

Scroll V: The O.W.L. Exam (A Capstone Project)

The Story (Docs - Left Side):

"Student... you are ready," Professor Flitwick says, his voice full of pride. "You've learned to store magic (Variables), use logic (Operators), write spells (Functions), organize your kit (Arrays/Objects), and even enchant the Living Parchment (The DOM)."

"But Lord Voldenull has sent a **Dementor-Bug** to the grounds! It's feeding on the page's happiness."

"Your final trial is here. Combine *all* your skills. We've given you the parchment for a small 'game.' Your quest is to bring it to life, cast your Patronus, and defeat the Dementor. Earn your first **Badge of Completion**!"

Your Quest (IDE - Right Side):

Your goal: Create a simple "Dementor's Duel" game.

The (hidden) HTML has:

- An <img> with id="dementor-image"
- An <h1> with id="health-display" that says "100"
- A <button> with id="patronus-button" that says "Cast Patronus!"

Your tasks:

1. Create a **variable** dementorHealth and set it to 100.
2. **Find** all three elements (the image, the h1, and the button) and store them in variables.
3. Create a **function** called castPatronus.
4. Inside castPatronus:
    - Subtract 10 from the dementorHealth variable.
    - Update the textContent of the health display <h1> to show the new dementorHealth.
    - Add an if statement: if (dementorHealth <= 0), then...
        - Set the Dementor image's style.display to "none" (to make it disappear).
        - Change the health display's textContent to "DEMENTOR DEFEATED!".
        - Change the health display's style.color to "lightblue".
5. **Add an event listener** to the patronus-button to run your castPatronus function on "click".
6. Go, Wizard! Click that button and save the school!

(This capstone quest combines everything: variables, operators, functions, DOM selection, DOM modification, and event listeners, all in one magical, game-like challenge.)