Cookie Clicker

Event-driven programming in JavaScript

Chris Dalziel and Adam Chapman-Dodd,
with design help from Moth Dalziel

What is Cookie Clicker?

Cookie Clicker is a web-based incremental video game written in JavaScript by French developer Julien “Orteil” Thiennot

Screenshot from https://en.wikipedia.org/wiki/Cookie_Clicker

What is Cookie Clicker?

The game has a simple progression loop revolving around the collection and spending of cookies, which serve both as points and as a kind of currency.

Cookies can be spent to purchase upgrades and auto-clickers, which in turn allows for the collection of more cookies.

Try Cookie Clicker!

Try it at https://orteil.dashnet.org/cookieclicker/ for five-ish minutes

What are the most important parts of the game? How might you implement these?

Event-based Programming

An **event** is an interaction between the user and the webpage

Event-based programming is a kind of reactive “When x do y” approach, which allows web-pages to react to the actions of the user dynamically

Event	Description
click	User clicks an element
mouseover	User moves the mouse over an element
onmouseout	User moves the mouse off an element
keydown	User presses a key

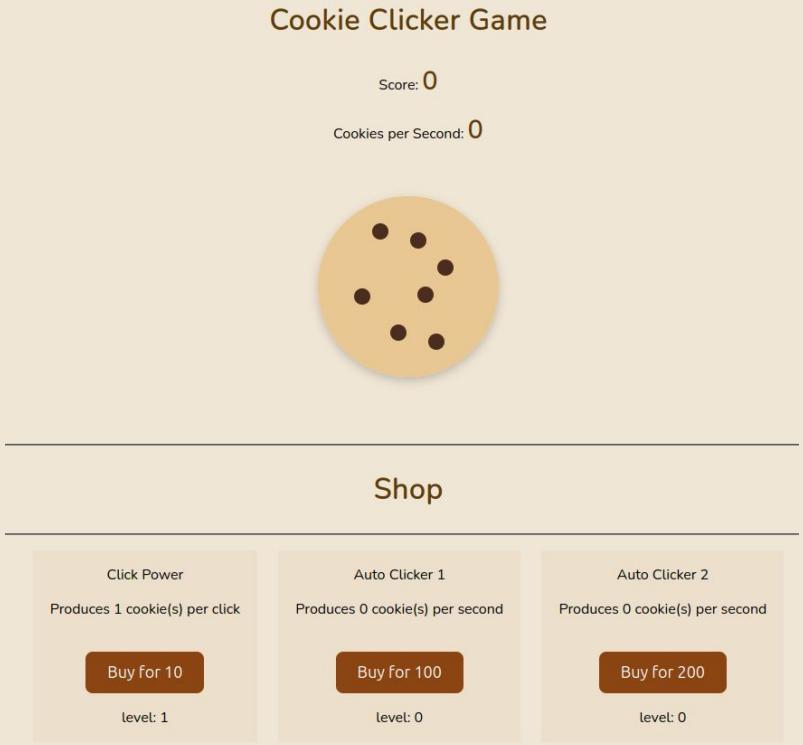
There are loads of events, not just these: https://www.w3schools.com/jsref/dom_obj_event.asp

Event-based Programming

Event-based approaches are also used in designing video games, where user interactions are of central importance

We can use event programming and the “click” event to build our own Cookie Clicker!

The provided project file contains an incomplete implementation of a Cookie Clicker game. Some of the code is missing, and there are bugs which need to be addressed



`id` and `class` in HTML

`id` and `class` are HTML attributes which can be set for any given element tag, and allow us to set properties for those elements. A `class` can be given to many elements, where an `id` is unique (and therefore identifying)

At National 5 we use these purely to set the styles of elements with CSS, but they can also be used with JavaScript to create “event handlers” for an element

Event Listeners - Learn

An **event listener** is a piece of code which waits for an event to happen, and runs function whenever it does

In this example, the code which updates the user's score runs whenever the cookie is clicked

```
// ——═══ Cookie clicking ═══——
let cookie = document.getElementById('cookie');
let scoreDisplay = document.getElementById('score');

// Event listener for cookie click
// Increment score on click
function cookieClick() {
    score += 1;
    refreshCookieCount();
}

cookie.addEventListener('click', cookieClick);

let refreshCookieCount = function() {
    scoreDisplay.textContent = score;
};
```

Aside — Functions as Arguments

JavaScript allows for functions themselves to be passed as an argument to other functions

This allows the function to be called from within these other functions, rather than just passing the result of these functions

This is useful for event listeners because we don’t necessarily want to call the function immediately when the page loads

```
// Passes the cookieClicked function to be called on click
cookie.addEventListener('click', cookieClicked);

// Calls cookieClicked on page load, passing the return
// value of the function (null) to the event listener
cookie.addEventListener('click', cookieClicked())
```

Event Listeners - Do

Your first task is to implement the most important feature — a clickable cookie!

Event Listeners - Discuss

- Why do we separate logic (JS), styling (CSS) and structure (HTML) rather than keep it all in the same file?
- How might you use different types of events (like `mouseenter`, `dblclick` or `keydown`) to change how players interact with this game?
- Why is `refreshCookieCount()` called after increasing the score? What would happen if we forgot to call it?
- How could you make this click interaction more engaging or apparent for a user?

Event Listeners - Summary

Periodic Functions - Learn

One “event” we might want to react to is the passage of a set amount of time

The “`setInterval`” function lets us run a function every time the provided interval has passed, allowing for periodically repeating functions

Periodic Functions - Learn

This `clock()` function increments the number of seconds that have passed and then adjusts the minutes and hours

For this to work correctly, we must call the `clock()` function exactly every second, which we do by passing the function and 1000 milliseconds to `setInterval`

Similarly to passing functions earlier we *don't* put brackets on the function name in the `setInterval` call

```
seconds = 0;
minutes = 0;
hours = 0;

function clock() {
    seconds++;
    if (seconds ⩾ 60) {
        seconds = 0;
        minutes++;
    }
    if (minutes ⩾ 60) {
        minutes = 0;
        hours++;
    }
    if (hours ⩾ 24) {
        hours = 0;
    }
    console.log(`${hours}:${minutes}:${seconds}`);
}

setInterval(clock, 1000)
```

Periodic Functions - Do

Now you should look at fixing the cookies per second counter, using the `setInterval` function to update it every second.

Periodic Functions - Discuss

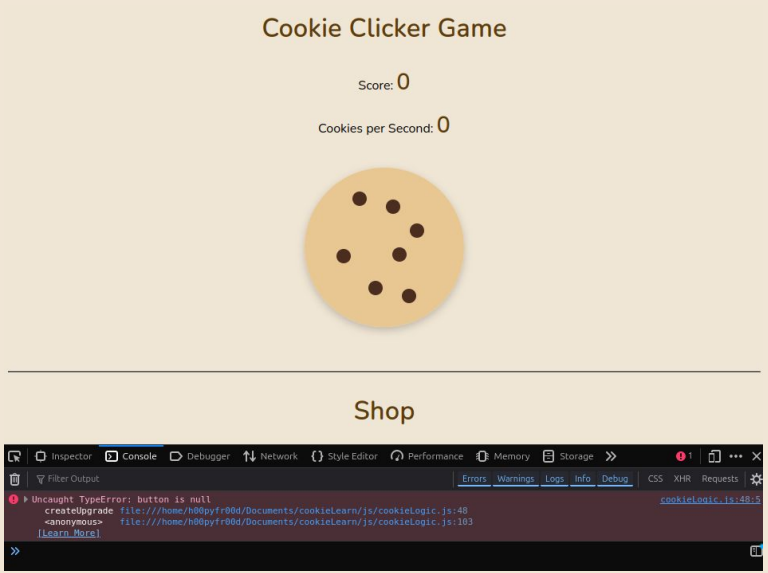
- Why use `setInterval()` instead of updating CPS on each click? What problems could occur if CPS was updated on every frame?

- Why do we need to reset the counter each second? What happens if you don’t do this?

Aside — Console debugging

The console tab of the debugger (accessible via Ctrl+Shift+K) can be very useful for finding bugs, as it shows any errors or warnings produced by your code

Additionally you can print your own messages to the console using `console.log()`



Aside — JavaScript Object Notation (JSON)

JavaScript's `Object` type is very versatile, as it allows us to collect many values into one object.

JavaScript Object Notation (more commonly abbreviated to JSON) lets us define the properties and values of an object using key-value syntax. The keys must be strings (quotes may be omitted), but the value may be any type — even functions and other objects

```
let object = {
    "key1": "value1",
    "key2": "value2",
     "key3": "value3"
}
```

```
let object2 = {
    "key1": 1,
    "key2": object,
    "key3": function() {
        console.log("Hello");
    }
}
```

Aside — JavaScript Object Notation (JSON)

We use JSON in the `createUpgrade` function to make it clearer what each argument is meant to be when calling the function

```
// Upgrade click power, making each click worth more cookies
createUpgrade({
    buttonId: 'buy-click-power',
    priceElementId: 'click-power-price',
    levelElementId: 'click-power-level',
    multipleElementId: 'click-power-multiple',
    priceAmount: 1,
    scalingAmount: 1,
    levelNumber: 1,
    power: (level) => 1,
    onUpgrade: function(level) {
        clickPower = 1;
    }
});
```

Timed Functions - Learn

Sometimes we'd like to time a function to happen some interval in the future, but not necessarily periodically

This example shows a secret message 10 seconds after the page loads. We only want to show the message once, so we use a timed function instead with `setTimeout`

```
function showSecretMessage() {
    showMessage("Secret message!");
}

setTimeout(showSecretMessage, 10000);
```

Timed Functions - Do

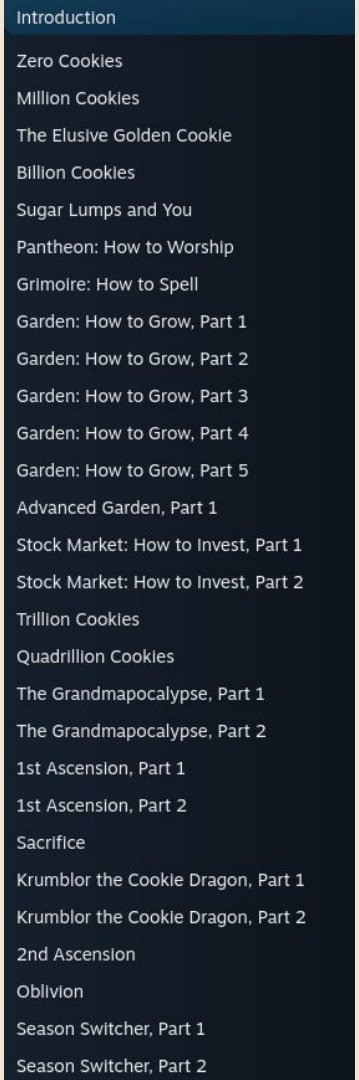
Timed Functions - Discuss

- Why is it a good idea to have one function like `createUpgrade()` handle upgrade setup?
- What is the issue with upgrade prices not scaling with level?
- How do HTML element IDs help JavaScript interact with the correct parts of a web page? What would happen if IDs are reused?

Where would you take it next?

Cookie Clicker has years of content — just look at this portion of a beginners Steam community guide!

Some of these mechanics are super out there, the potential for new ones is limitless: what would you add?



https://steamcommunity.com/sharedfiles/filedetails/?id=2591407770

Summary

- **Events** are user interactions with the page
- We can react to events dynamically using **event listeners**
- If we want to **delay** the call of a function we can use `setInterval` and `setTimeout`, where an interval function will repeat periodically and a timeout function will run once (unless called within itself!)