Working with Numbers, Dates, and Timers in JavaScript

This learning material explores JavaScript's powerful tools for working with numbers, dates, and timers. Each section includes comprehensive explanations, practical examples, and common use cases.

---

2. Math and Rounding (Expanded)

The `Math` object in JavaScript provides a variety of built-in mathematical methods and constants to handle calculations, rounding, and random number generation.

Key Math Constants

- `Math.PI`: The value of π (approximately 3.14159).
- `Math.E`: The base of the natural logarithm (Euler's number).

Example:

```
console.log(Math.PI); // Output: 3.141592653589793
console.log(Math.E); // Output: 2.718281828459045
```

Math Functions

The `Math` object provides numerous utility methods for calculations:

1. **Square Roots and Exponents**:

   ```
   console.log(Math.sqrt(16)); // Output: 4
   console.log(8 ** (1 / 3)); // Output: 2 (cube root)
   console.log(3 ** 4); // Output: 81 (3 to the power of 4)
   ```

2. **Finding Min and Max**:
   These methods can handle multiple arguments. Note that `Math.max` and `Math.min` will ignore non-numeric values unless they can be converted to numbers.

   ```
   console.log(Math.max(10, 20, 5)); // Output: 20
   console.log(Math.min(10, 20, "15")); // Output: 10
   console.log(Math.max(10, "30px", 5)); // Output: NaN
   ```

3. **Random Numbers**:
   `Math.random()` generates a random number between `0` (inclusive) and `1` (exclusive). For custom ranges, you can use:

```
const randomInt = (min, max) =>
  Math.floor(Math.random() * (max - min + 1)) + min;

console.log(randomInt(1, 10)); // Random number between 1 and 10
```

Rounding Numbers

JavaScript provides various methods to round numbers to integers:

- `Math.round()`: Rounds to the nearest integer.
- `Math.ceil()`: Rounds up to the next integer.
- `Math.floor()`: Rounds down to the nearest integer.
- `Math.trunc()`: Removes the fractional part of a number.

```
console.log(Math.round(23.5)); // Output: 24
console.log(Math.ceil(23.1)); // Output: 24
console.log(Math.floor(23.9)); // Output: 23
console.log(Math.trunc(23.9)); // Output: 23
```

Special Case: Negative numbers behave differently for `Math.trunc` and `Math.floor`:

```
console.log(Math.trunc(-23.5)); // Output: -23
console.log(Math.floor(-23.5)); // Output: -24
```

Rounding Decimals:
To round decimals to a specific number of places, use `Number.prototype.toFixed()`:

```
console.log((2.345).toFixed(2)); // Output: "2.35" (as a string)
console.log(+(2.345).toFixed(2)); // Output: 2.35 (as a number)
```

---

3. The Remainder Operator

The `%` operator computes the remainder of a division operation. It is particularly useful for:

- Checking even or odd numbers.
- Implementing cyclical behavior.

Examples:

```
console.log(7 % 2); // Output: 1 (7 divided by 2 has a remainder of 1)
console.log(6 % 2); // Output: 0 (6 is divisible by 2)
```

Practical Use Case: Highlighting every second or third element in a list.

```
const elements = [...document.querySelectorAll(".list-item")];
elements.forEach((el, index) => {
  if (index % 2 === 0) el.style.backgroundColor = "lightblue"; // Every 2nd
item
  if (index % 3 === 0) el.style.backgroundColor = "lightgreen"; // Every 3rd
item
});
```

---

4. Numeric Separators

Numeric separators (_) improve code readability when working with large numbers, without affecting their value. They can be used in numbers, but not in strings.

Examples:

```
const budget = 1_000_000; // One million
console.log(budget); // Output: 1000000

const speed = 300_000_000; // Speed of light in m/s
console.log(speed); // Output: 300000000
```

Caution:

- Numeric separators are not supported in string conversions.

```
console.log(Number("100_000")); // Output: NaN
```

---

5. Working with BigInt (Expanded)

`BigInt` is a special numeric type in JavaScript introduced to handle integers larger than `Number.MAX_SAFE_INTEGER` (2^53 - 1). It can safely represent and perform calculations on extremely large integers.

Key Features of BigInt

- **Creation**: BigInt values are created by appending an `n` to an integer literal (e.g., `123n`) or by using the `BigInt()` constructor.
- **Safety**: Regular numbers lose precision beyond `Number.MAX_SAFE_INTEGER`. BigInt ensures precision for very large numbers.
- **Operations**: BigInt supports basic arithmetic like addition, subtraction, multiplication, and exponentiation.
- **Mixing with Regular Numbers**: BigInt cannot be directly mixed with regular numbers in operations. Explicit conversion is required.

Examples

1. **Creating BigInt Values**:

```
// Using 'n' suffix
const bigInt1 = 12345678901234567890123456789012345678901234567890n;

// Using BigInt constructor
const bigInt2 = BigInt("12345678901234567890123456789012345678901234567890");

console.log(bigInt1 === bigInt2); // Output: true
```

2. **Arithmetic Operations**:

```
const bigIntA = 2n ** 53n; // 2 to the power of 53
console.log(bigIntA); // Output: 9007199254740992n

const bigIntB = BigInt(9007199254740991) + 1n;
console.log(bigIntB); // Output: 9007199254740992n
```

3. **Mixing BigInt and Regular Numbers**:

```
const num = 42;
const bigIntC = 12345678901234567890n;

// Explicit conversion is required
console.log(bigIntC + BigInt(num)); // Works
console.log(Number(bigIntC) + num); // Works (but loses precision for
large BigInt values)

// Implicit mixing throws an error
// console.log(bigIntC + num); // TypeError
```

4. **Comparison**:
   BigInt and regular numbers can be compared directly (no explicit conversion needed):

```
console.log(100n > 50); // Output: true
console.log(100n === 100); // Output: false (different types)
console.log(100n == 100); // Output: true (type coercion)
```

5. **Edge Cases**:
   BigInt doesn't support decimals:

```
// Invalid operation
// console.log(10.5n); // SyntaxError
```

---

6. Dates in JavaScript (Expanded)

JavaScript's `Date` object provides powerful utilities to work with dates and times.

Creating Dates

1. **Current Date and Time**:

```
const now = new Date();
console.log(now); // Output: Current date and time
```

2. **Specific Date**:

```
const specificDate = new Date("2025-01-26T10:30:00");
console.log(specificDate); // Output: Sun Jan 26 2025 10:30:00 GMT+0000
(UTC)
```

3. **Custom Date (Year, Month, Day, Hour, Minute)**:

```
const customDate = new Date(2037, 10, 19, 15, 23, 5);
// Month is 0-indexed: 10 = November
console.log(customDate); // Output: Thu Nov 19 2037 15:23:05 GMT+0000
(UTC)
```

4. **Timestamps**:
   A timestamp represents the number of milliseconds since January 1, 1970 (Unix epoch).

```
const timestamp = Date.now();
console.log(timestamp); // Output: Current timestamp
```

Date Methods

1. **Get Components**:

```
const date = new Date(2037, 10, 19, 15, 23);
console.log(date.getFullYear()); // Output: 2037
console.log(date.getMonth()); // Output: 10 (November)
console.log(date.getDate()); // Output: 19
console.log(date.getDay()); // Output: 4 (Thursday)
console.log(date.getHours()); // Output: 15
console.log(date.getMinutes()); // Output: 23
console.log(date.toISOString()); // Output: ISO 8601 format
```

2. **Set Components**:

```
const date = new Date(2037, 10, 19);
date.setFullYear(2040);
console.log(date.getFullYear()); // Output: 2040
```

Date Calculations

Perform arithmetic with dates to calculate time differences:

```
const date1 = new Date(2037, 3, 10);
const date2 = new Date(2037, 3, 20);

const diffTime = date2 - date1; // Difference in milliseconds
const diffDays = diffTime / (1000 * 60 * 60 * 24); // Convert to days
console.log(diffDays); // Output: 10
```

Working with Time Zones

JavaScript's `Date` object includes built-in time zone support:

```
const now = new Date();
console.log(now.getTimezoneOffset()); // Output: Time zone offset in minutes
```

Use libraries like `date-fns` or `Moment.js` for advanced time zone handling.

---

7. Internationalizing Numbers (Expanded)

The `Intl.NumberFormat` API in JavaScript allows you to format numbers based on specific locales and options, making it easier to present numbers in different cultural formats.

Key Concepts of `Intl.NumberFormat`

1. **Locale**:
   - Specifies the language and region to format numbers.
   - Examples: `"en-US"` (English, United States), `"de-DE"` (German, Germany), `"ja-JP"` (Japanese, Japan).
2. **Options**:
   - Define how the number should be formatted (e.g., style, currency, unit).

Examples:

1. **Basic Number Formatting**:

```
const number = 1234567.89;

console.log(new Intl.NumberFormat("en-US").format(number)); // Output:
1,234,567.89
console.log(new Intl.NumberFormat("de-DE").format(number)); // Output:
1.234.567,89
console.log(new Intl.NumberFormat("ja-JP").format(number)); // Output:
1,234,567.89
```

2. **Currency Formatting**:

```
const currencyOptions = { style: "currency", currency: "USD" };
console.log(new Intl.NumberFormat("en-US",
currencyOptions).format(1234.56));
// Output: $1,234.56

console.log(
  new Intl.NumberFormat("de-DE", {
    style: "currency",
    currency: "EUR",
  }).format(1234.56)
);
```

```
// Output: 1.234,56 €

console.log(
  new Intl.NumberFormat("ja-JP", {
    style: "currency",
    currency: "JPY",
  }).format(1234)
);
// Output: ￥1,234
```

3. **Percentage Formatting**:

```
const percentOptions = { style: "percent", minimumFractionDigits: 2 };
console.log(new Intl.NumberFormat("en-US",
percentOptions).format(0.12345));
// Output: 12.35%

console.log(new Intl.NumberFormat("de-DE",
percentOptions).format(0.12345));
// Output: 12,35 %
```

4. **Unit Formatting**:

```
const unitOptions = { style: "unit", unit: "kilometer-per-hour" };
console.log(new Intl.NumberFormat("en-US", unitOptions).format(90));
// Output: 90 km/h
```

5. **Customizing Decimal Places**:

```
const numberOptions = { minimumFractionDigits: 2, maximumFractionDigits:
3 };
console.log(
  new Intl.NumberFormat("en-US", numberOptions).format(1234.56789)
);
// Output: 1,234.568
```

---

8. Timers (Expanded)

Timers are crucial in JavaScript for scheduling tasks. The two primary functions for timers are `setTimeout` and `setInterval`.

---

setTimeout: Delayed Execution

- Executes a function once after a specified delay (in milliseconds).

```
setTimeout(() => console.log("Hello after 3 seconds!"), 3000);
// Output (after 3 seconds): Hello after 3 seconds!
```

1. **Passing Arguments**:

```
const greet = (name) => console.log(`Hello, ${name}!`);
setTimeout(greet, 2000, "John");
// Output (after 2 seconds): Hello, John!
```

2. **Clearing a Timeout**:

```
const timer = setTimeout(() => console.log("This will not run"), 5000);
clearTimeout(timer); // Cancels the timeout
```

---

setInterval: Repeated Execution

- Executes a function repeatedly at a specified interval (in milliseconds).

```
setInterval(() => console.log("Tick"), 1000);
// Output: "Tick" printed every second
```

1. **Stopping** `setInterval`:

```
let counter = 0;
const interval = setInterval(() => {
  counter++;
  console.log(`Counter: ${counter}`);
  if (counter === 5) clearInterval(interval); // Stops after 5
repetitions
}, 1000);
```

Practical Timer Examples

1. **Countdown Timer**:

```
let timeLeft = 10;

const countdown = setInterval(() => {
  console.log(`Time left: ${timeLeft}s`);
  timeLeft--;
  if (timeLeft < 0) {
    clearInterval(countdown);
    console.log("Countdown complete!");
  }
}, 1000);
```

2. **Real-Time Clock**:

```
setInterval(() => {
  const now = new Date();
  console.log(now.toLocaleTimeString());
}, 1000);
```

Challenge Solution

1. **Random Number Generator with Currency Formatting and Timer**:

```
const randomCurrency = (min, max, locale, currency) => {
  const randomNum = Math.random() * (max - min) + min;
  const formatted = new Intl.NumberFormat(locale, {
    style: "currency",
    currency: currency,
  }).format(randomNum);

  setTimeout(() => {
    console.log(`Random number: ${formatted}`);
  }, 2000);
};

randomCurrency(100, 1000, "en-US", "USD");
// Output (after 2 seconds): Random number: $423.87
```

2. **Days Until New Year's**:

```
const daysUntilNewYear = () => {
  const now = new Date();
  const nextYear = now.getFullYear() + 1;
  const newYear = new Date(nextYear, 0, 1);
  const diffTime = newYear - now;
```

```
  const diffDays = Math.ceil(diffTime / (1000 * 60 * 60 * 24));
  console.log(`Days until New Year: ${diffDays}`);
};

daysUntilNewYear();
```
