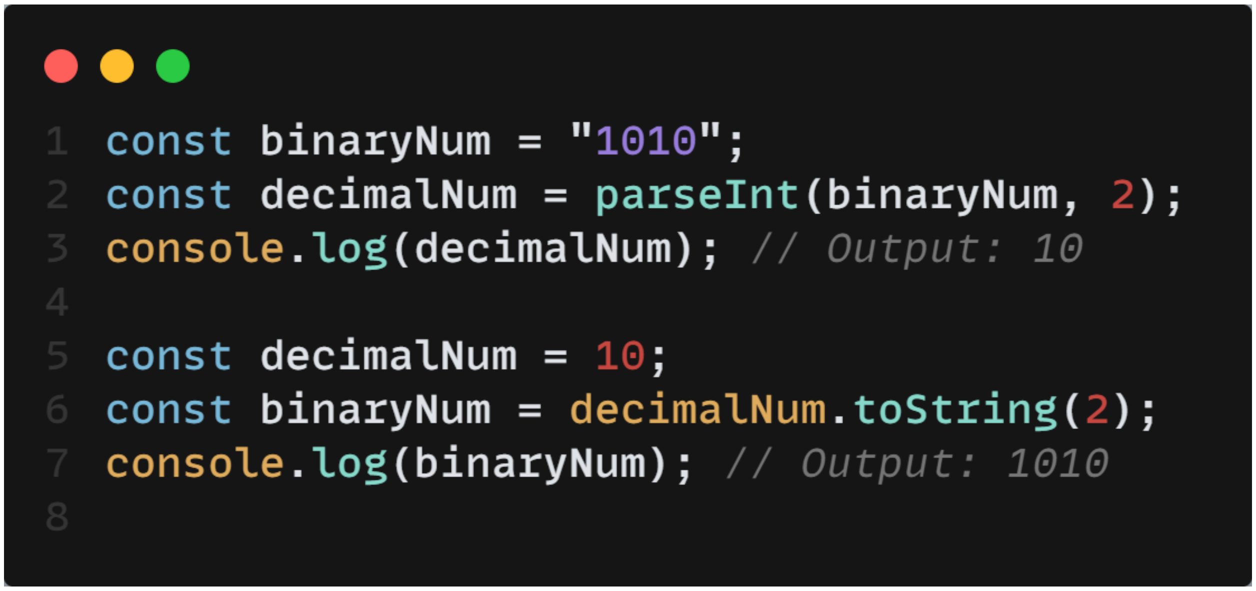
BIT MANIPULATION

Bit manipulation is the process of manipulating individual bits of a binary number. It is a technique used in computer programming to perform operations at the bit level. Bit manipulation is often used in low-level programming, such as device drivers and embedded systems.



```
const binaryNum = "1010";
const decimalNum = parseInt(binaryNum, 2);
console.log(decimalNum); // Output: 10

const decimalNum = 10;
const binaryNum = decimalNum.toString(2);
console.log(binaryNum); // Output: 1010

```

Here are some examples of bit manipulation:

1. Bitwise AND (&): This operator compares two bits and returns 1 if both bits are 1, otherwise it returns 0. For example, 5 & 3 would be:

```
   ```
   0101 (5 in binary)
& 0011 (3 in binary)

```
```

```
 0001 (1 in binary)
```

2. Bitwise OR (|): This operator compares two bits and returns 1 if either bit is 1, otherwise it returns 0. For example, 5 | 3 would be:

```
  0101 (5 in binary)
| 0011 (3 in binary)
  ------
  0111 (7 in binary)
```

3. Bitwise XOR (^): This operator compares two bits and returns 1 if the bits are different, otherwise it returns 0. For example, 5 ^ 3 would be:

```
  0101 (5 in binary)
^ 0011 (3 in binary)
  -----
  0110 (6 in binary)
```

4. Bitwise NOT (~): This operator flips the bits of a number. For example, ~5 would be:

```
0101 (5 in binary)
------
1010 (10 in binary)
```

The sign bit is a bit in a binary number that represents the sign of a signed integer. It is typically the leftmost bit in a binary representation, where 0 represents a positive number and 1 represents a negative number.

In signed integer representations, such as two's complement, the sign bit is used to indicate the sign of the number. For example, in an 8-bit signed integer representation, the sign bit is the leftmost bit (bit 7), and the remaining 7 bits represent the magnitude of the number.

If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative. The remaining bits in the binary representation determine the magnitude of the number.

For example, in an 8-bit signed integer representation:

- The binary number `01100101` represents the positive number 101.
- The binary number `11100101` represents the negative number -101.

The sign bit is important for arithmetic operations and determining the range of values that can be represented in a signed integer representation. It allows for the representation of both positive and negative numbers using a fixed number of bits.

Left and right shift are bitwise operators used in bit manipulation. They are used to shift the bits of a binary number to the left or right.

Left shift (<<) is an operator that shifts the bits of a binary number to the left by a specified number of positions. The leftmost bits are filled with zeros. For example, if we have the binary number 1010 and we left shift it by 2 positions, we get:

```
1010 << 2 = 101000
```

Right shift (>>) is an operator that shifts the bits of a binary number to the right by a specified number of positions. The rightmost bits are filled with zeros. For example, if we have the binary number 1010 and we right shift it by 2 positions, we get:

```
1010 >> 2 = 0010
```

Right shift can also be used with a sign bit to preserve the sign of a number. This is called arithmetic right shift. In this case, the sign bit is shifted along with the other bits. For example, if we have the binary number 1101 and we arithmetic right shift it by 1 position, we get:

```
1101 >> 1 = 1110
```

In this case, the leftmost bit (1) is the sign bit, and it is preserved during the shift.

Left shift and right shift are bitwise operators used in bit manipulation. They are used to shift the bits of a binary number to the left or right.

Left shift (<<) is an operator that shifts the bits of a binary number to the left by a specified number of positions. The leftmost bits are filled with zeros. For example, if we have the binary number 1010 and we left shift it by 2 positions, we get:

```
1010 << 2 = 101000
```

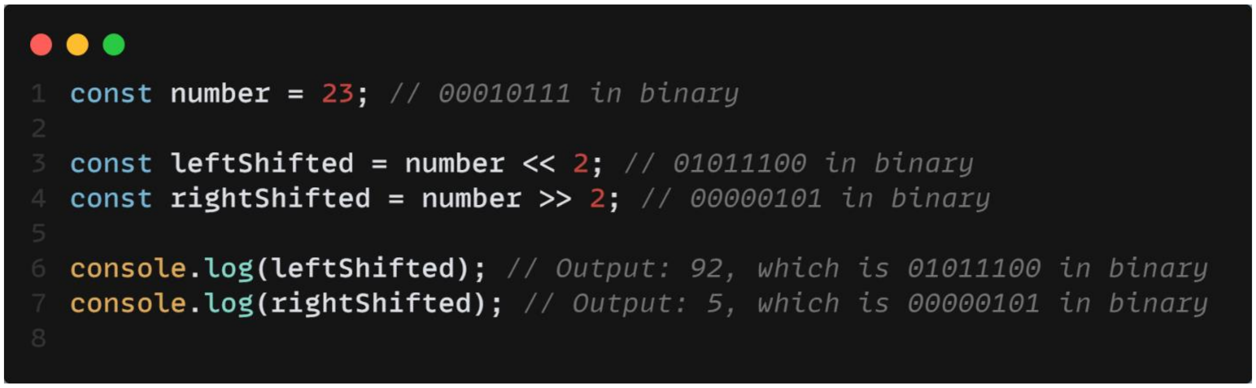
Right shift (>>) is an operator that shifts the bits of a binary number to the right by a specified number of positions. The rightmost bits are filled with zeros. For example, if we have the binary number 1010 and we right shift it by 2 positions, we get:

```
1010 >> 2 = 0010
```

Right shift can also be used with a sign bit to preserve the sign of a number. This is called arithmetic right shift. In this case, the sign bit is shifted along with the other bits. For example, if we have the binary number 1101 and we arithmetic right shift it by 1 position, we get:

```
1101 >> 1 = 1110
```

In this case, the leftmost bit (1) is the sign bit, and it is preserved during the shift.



```
const number = 23; // 00010111 in binary

const leftShifted = number << 2; // 01011100 in binary
const rightShifted = number >> 2; // 00000101 in binary

console.log(leftShifted); // Output: 92, which is 01011100 in binary
console.log(rightShifted); // Output: 5, which is 00000101 in binary

```

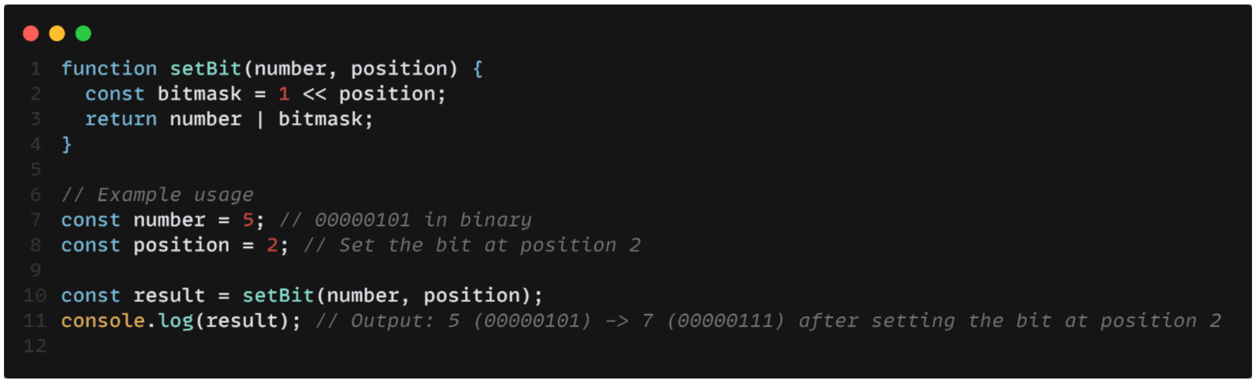
In the above example, the number variable is set to 23, which is 00010111 in binary. The leftShifted variable is set to the result of shifting the number variable to the left by 2 bits using the left shift operator (<<). This operation adds two 0 bits to the right of the binary number, resulting in 01011100 in binary, which is 92 in decimal.

The rightShifted variable is set to the result of shifting the number variable to the right by 2 bits using the right shift operator (>>). This operation removes two bits from the right of the binary number, resulting in 00000101 in binary, which is 5 in decimal.

Note that the left shift operator (<<) adds 0 bits to the right of the binary number, while the right shift operator (>>) adds 0 bits to the left of the binary number if the number is positive, and 1 bits to the left if the number is negative.

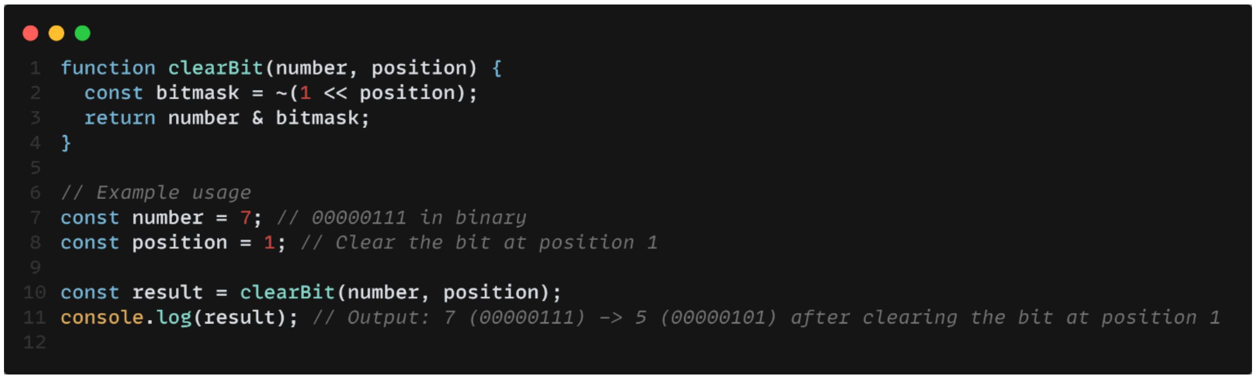
- **Set a bit in a number**
    - Create a bitmask: Create a bitmask with a 1 at the desired bit position and 0s everywhere else. You can do this by left shifting 1 by the desired bit position.

- Perform a bitwise OR operation: Use the bitwise OR operator (`|`) to perform a bitwise OR operation between the bitmask and the original number. This will set the bit at the desired position to 1 while leaving other bits unchanged.



```javascript
function setBit(number, position) {
  const bitmask = 1 << position;
  return number | bitmask;
}

// Example usage
const number = 5; // 00000101 in binary
const position = 2; // Set the bit at position 2

const result = setBit(number, position);
console.log(result); // Output: 5 (00000101) -> 7 (00000111) after setting the bit at position 2

```

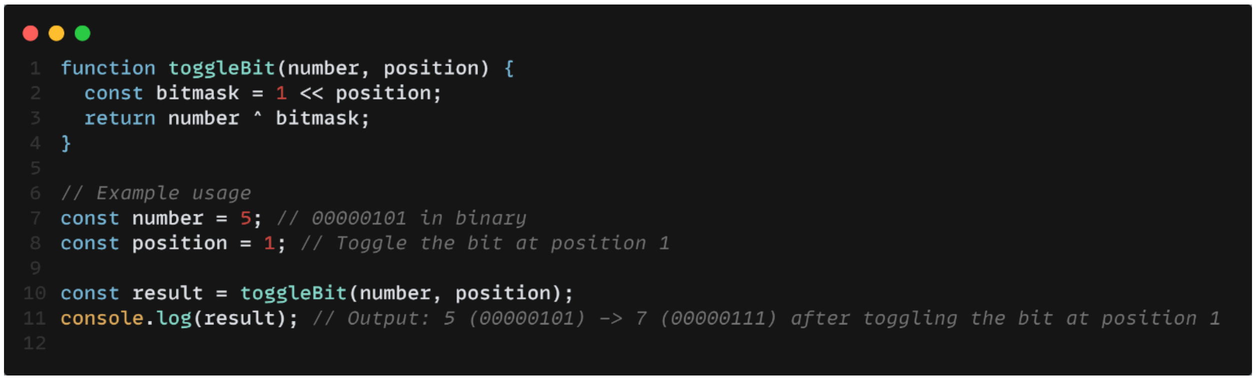
- **Clear a bit in a number**
    - Create a bitmask: Create a bitmask with a 0 at the desired bit position and 1s everywhere else.
    - Perform a bitwise AND operation: Use the bitwise AND operator (`&`) to perform a bitwise AND operation between the bitmask (complemented) and the original number. This will clear the bit at the desired position while leaving other bits unchanged.



```javascript
function clearBit(number, position) {
  const bitmask = ~(1 << position);
  return number & bitmask;
}

// Example usage
const number = 7; // 00000111 in binary
const position = 1; // Clear the bit at position 1

const result = clearBit(number, position);
console.log(result); // Output: 7 (00000111) -> 5 (00000101) after clearing the bit at position 1

```

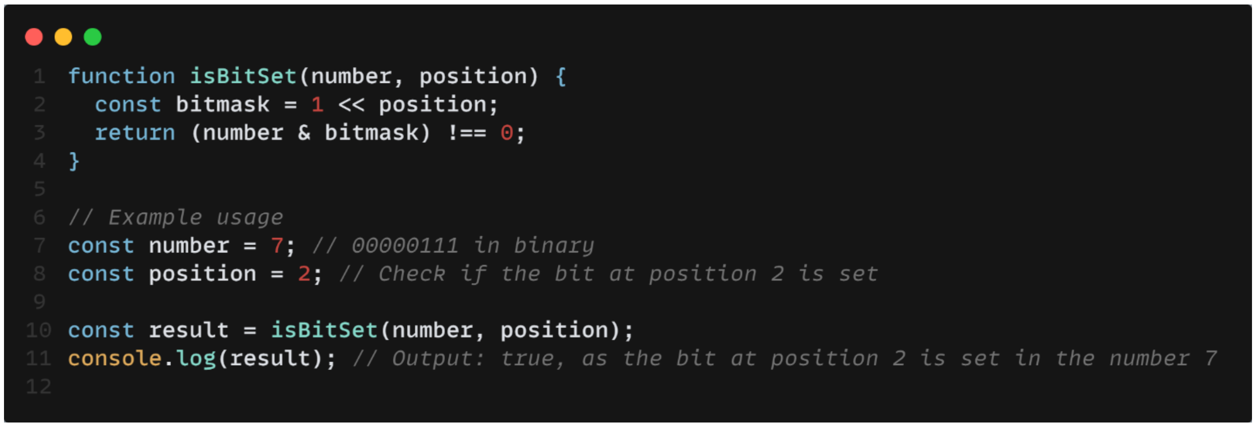
- **Toggle a bit in a number**

  - Create a bitmask: Create a bitmask with a 1 at the desired bit position and 0s everywhere else. You can do this by left shifting 1 by the desired bit position.
  - Perform a bitwise XOR operation: Use the bitwise XOR operator (`^`) to perform a bitwise XOR operation between the bitmask and the original number. This will toggle the bit at the desired position while leaving other bits unchanged.



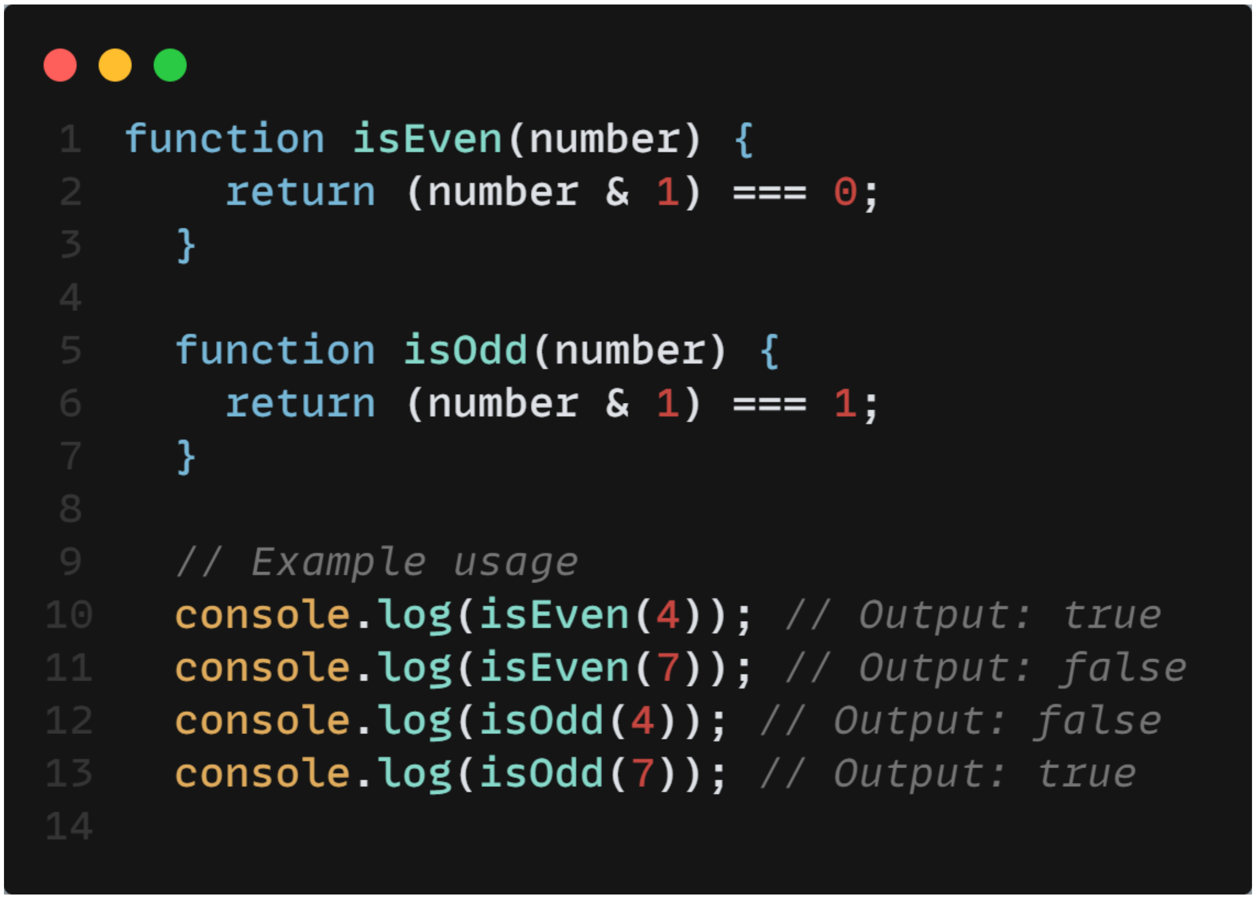
```javascript
function toggleBit(number, position) {
  const bitmask = 1 << position;
  return number ^ bitmask;
}

// Example usage
const number = 5; // 00000101 in binary
const position = 1; // Toggle the bit at position 1

const result = toggleBit(number, position);
console.log(result); // Output: 5 (00000101) -> 7 (00000111) after toggling the bit at position 1
```

- **Check if a bit is set**
  - Create a bitmask: Create a bitmask with a 1 at the desired bit position and 0s everywhere else. You can do this by left shifting 1 by the desired bit position.
  - Perform a bitwise AND operation: Use the bitwise AND operator (`&`) to perform a bitwise AND operation between the bitmask and the original number. If the result is non-zero, it means the bit at the desired position is set.



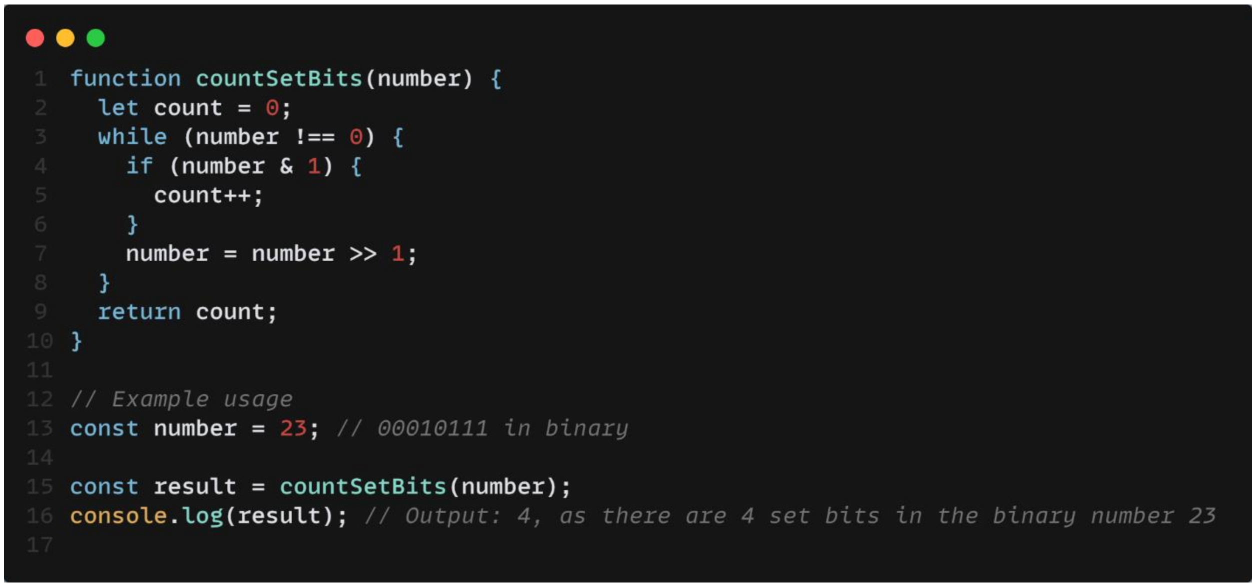
```javascript
function isBitSet(number, position) {
  const bitmask = 1 << position;
  return (number & bitmask) !== 0;
}

// Example usage
const number = 7; // 00000111 in binary
const position = 2; // Check if the bit at position 2 is set

const result = isBitSet(number, position);
console.log(result); // Output: true, as the bit at position 2 is set in the number 7
```

- **Check Even and Odd Number**



```javascript
function isEven(number) {
      return (number & 1) === 0;
    }

    function isOdd(number) {
      return (number & 1) === 1;
    }

    // Example usage
    console.log(isEven(4)); // Output: true
    console.log(isEven(7)); // Output: false
    console.log(isOdd(4)); // Output: false
    console.log(isOdd(7)); // Output: true
```

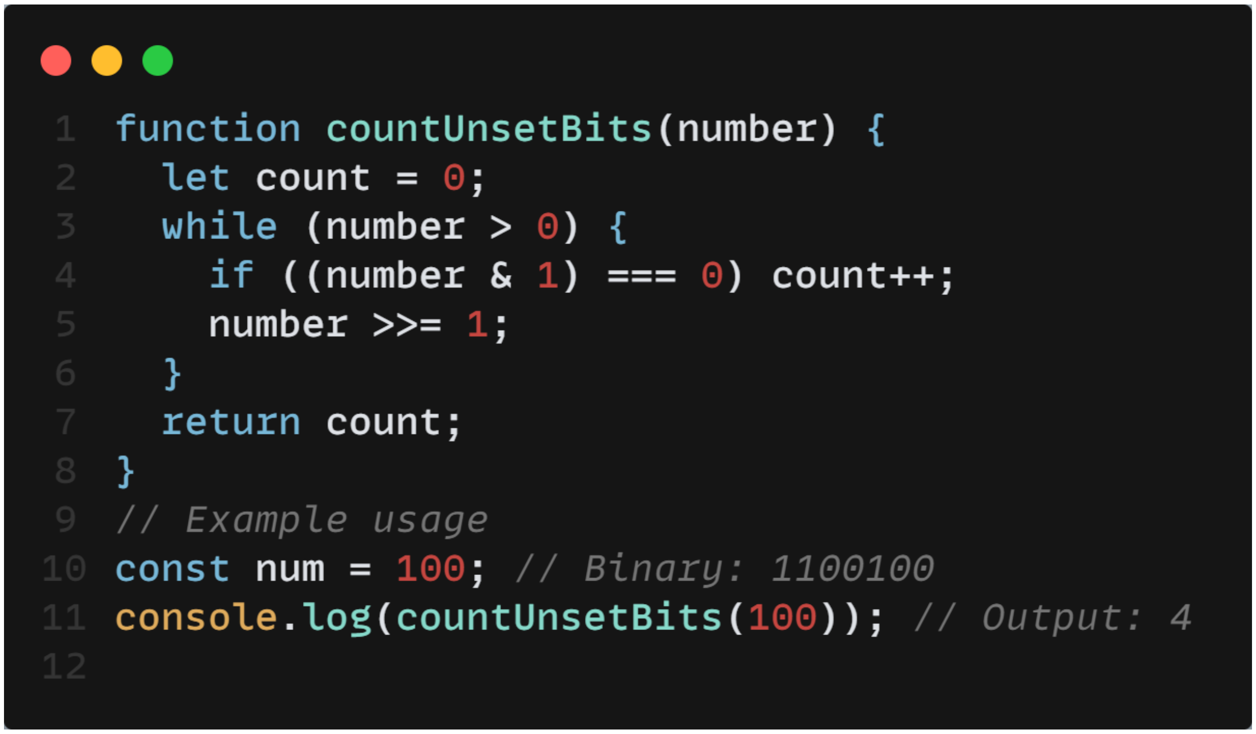
- **Count the number of set bits (bits with a value of 1)**
  - Initialize a count variable to 0.
  - Use a loop to iterate over each bit in the binary number. You can do this by shifting the number to the right by 1 bit each time and checking the value of the rightmost bit using the bitwise AND operator (`&`).
  - If the rightmost bit is 1, increment the count variable.
  - Repeat steps 2-3 until the entire binary number has been processed.
  - Return the count variable as the output.



```
function countSetBits(number) {
  let count = 0;
  while (number !== 0) {
    if (number & 1) {
      count++;
    }
    number = number >> 1;
  }
  return count;
}

// Example usage
const number = 23; // 00010111 in binary

const result = countSetBits(number);
console.log(result); // Output: 4, as there are 4 set bits in the binary number 23

```

- **Count the number of unset bits (bits with a value of 0)**
  - Initialize a count variable to 0.
  - Use a loop to iterate over each bit in the binary number. You can do this by shifting the number to the right by 1 bit each time and checking the value of the rightmost bit using the bitwise AND operator (`&`).
  - If the rightmost bit is 0, increment the count variable.

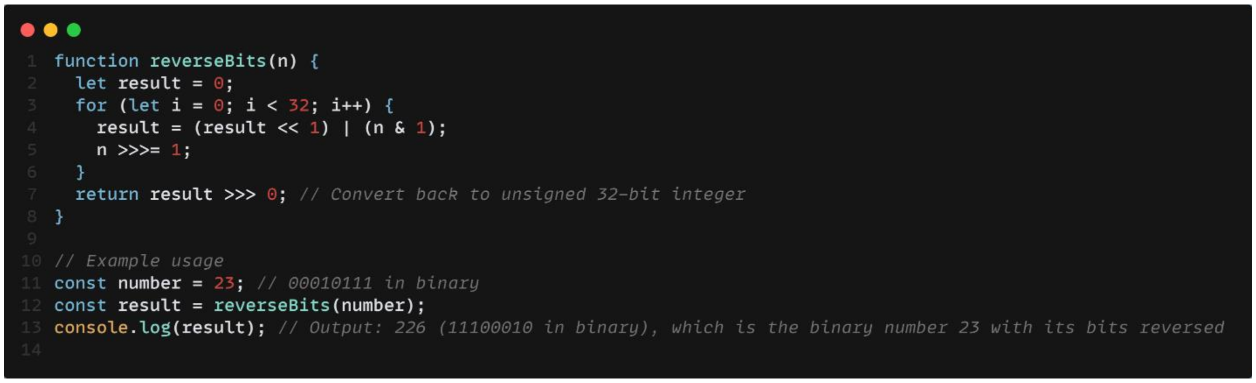
- Repeat steps 2-3 until the entire binary number has been processed.
- Return the count variable as the output.



```javascript
function countUnsetBits(number) {
  let count = 0;
  while (number > 0) {
    if ((number & 1) === 0) count++;
    number >>= 1;
  }
  return count;
}
// Example usage
const num = 100; // Binary: 1100100
console.log(countUnsetBits(100)); // Output: 4

```

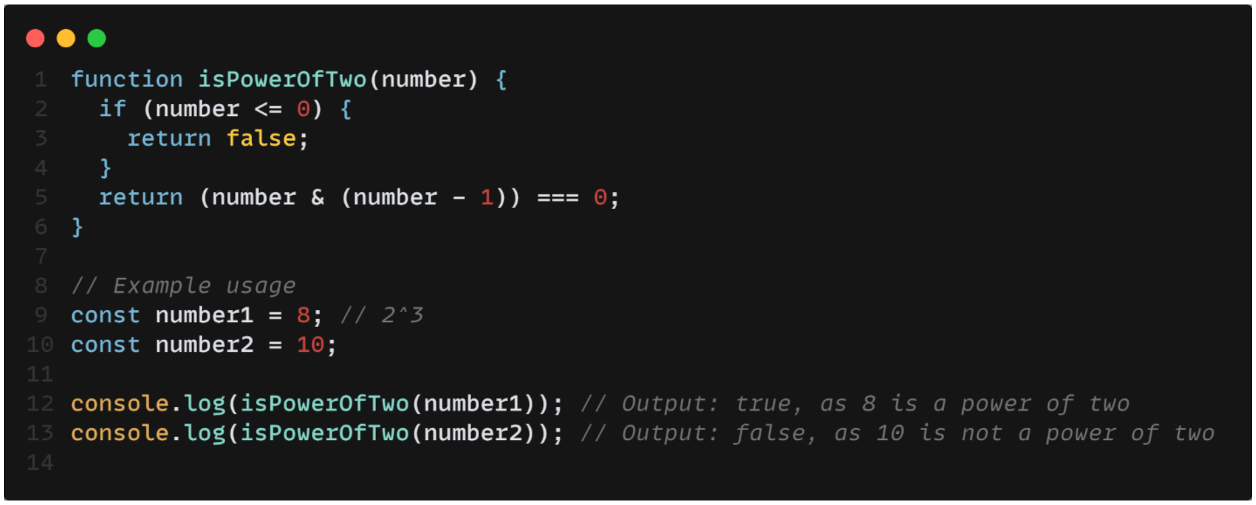
- **Reverse Bits**
  - Initialize a result variable to 0.
  - Use a loop to iterate over each bit in the binary number. You can do this by shifting the number to the right by 1 bit each time and checking the value of the rightmost bit using the bitwise AND operator (`&`).
  - If the rightmost bit is 1, set the corresponding bit in the result variable to 1 using the bitwise OR operator (`|`).
  - Shift the result variable to the left by 1 bit.
  - Repeat steps 2-4 until the entire binary number has been processed.

- Return the result variable as the output.



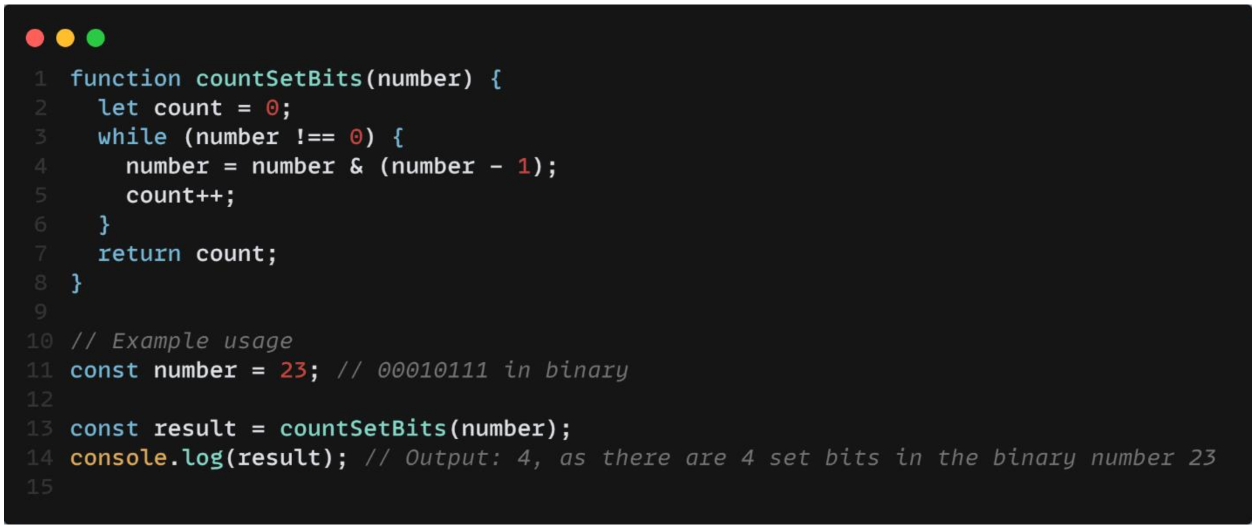
```javascript
function reverseBits(n) {
  let result = 0;
  for (let i = 0; i < 32; i++) {
    result = (result << 1) | (n & 1);
    n >>>= 1;
  }
  return result >>> 0; // Convert back to unsigned 32-bit integer
}

// Example usage
const number = 23; // 00010111 in binary
const result = reverseBits(number);
console.log(result); // Output: 226 (11100010 in binary), which is the binary number 23 with its bits reversed

```

- **Is Power of 2**
  - Check if the number is greater than 0.
  - Use the bitwise AND operator (`&`) to check if the number is a power of two. A number is a power of two if and only if it has only one bit set to 1. For example, 2 is a power of two because its binary representation is 10, which has only one bit set to 1. On the other hand, 3 is not a power of two because its binary representation is 11, which has two bits set to 1.
  - If the number is a power of two, the result of the bitwise AND operation between the number and its predecessor (i.e., the number minus 1) will be 0.



```javascript
function isPowerOfTwo(number) {
  if (number <= 0) {
    return false;
  }
  return (number & (number - 1)) === 0;
}

// Example usage
const number1 = 8; // 2^3
const number2 = 10;

console.log(isPowerOfTwo(number1)); // Output: true, as 8 is a power of two
console.log(isPowerOfTwo(number2)); // Output: false, as 10 is not a power of two
```

- **Brian Kernighan's Algorithm**
  Brian Kernighan's algorithm is used to find the number of set bits in a number. The idea behind the algorithm is that when we subtract one from an integer, all the bits following the rightmost set of bits are inverted, turning 1 to 0 and 0 to 1. The rightmost set bit also gets inverted with the bits right to it.



```javascript
function countSetBits(number) {
  let count = 0;
  while (number !== 0) {
    number = number & (number - 1);
    count++;
  }
  return count;
}

// Example usage
const number = 23; // 00010111 in binary

const result = countSetBits(number);
console.log(result); // Output: 4, as there are 4 set bits in the binary number 23
```