Module 3: Introduction to JavaScript

Video 3.5 Basic Data Types in JavaScript

The kind of data that a variable can contain in a programming language is called a data type. A data type defines the type of values that a variable can store and the operations that can be performed on those values. Data types are essential to programming because they ensure that data is used appropriately and effectively in a computer program.

Different programming languages may have their own unique data types. JavaScript is a dynamically typed programming language, which means that variables are not explicitly declared with a data type. They are determined at runtime.

The following is a list of some of the basic data types in JavaScript:

Number:

Number represents both integers and floating-point numbers. Integers are whole numbers and negative numbers without a decimal point. Floating-point numbers are decimal numbers that contain a decimal point. Number can be positive, negative, or zero. For example:

```
let num = 42; // number - integer
let pi = 3.14159; // number - float (floating point number)
```

String:

String represents a sequence of characters enclosed in single (') or double (") quotes. String is so named because it can be compared to a string of beads. Imagine a string that has a sequence of beads; similarly, the string data type has a sequence of characters. For example:

```
let name = "John"; // sequence of chars - 'J', 'o', 'h', 'n'
let message = 'Hello, World!';
```

Boolean:

Boolean represents a binary value: either true or false. This data type can be used for logical operations and conditions. For example:

```
let isTrue = true; // binary value - 1
let isFalse = false; // binary value - 0
```

Undefined:

Undefined is a variable that has not been defined. If you try to access a variable that has not yet been assigned a value, the variable has a data type of undefined. For example:

```
let uninitializedVar; // no value is defined, hence, undefined
console.log(uninitializedVar); // Outputs: undefined
```

Null:

Null represents the absence of any object value, or no value at all. For example:

```
let emptyValue = null;
```

Object

Object represents a collection of key-value pairs, where keys can be used to access their values from the object. Keys are strings and values can be of any type. An object is a great alternative when you need to define a few variables related to an entity and you need to keep them together. For example:

```
let person = {
    name: "Alice",
    age: 30,
    isStudent: false
};
```

You can access the values in person object by using either dot notation or square bracket notation. Using dot notation, you can access the values as follows:

```
// Accessing values using dot notation
console.log(person.name);       // "Alice"
console.log(person.age);        // 30
console.log(person.isStudent);  // false
```

Note that you can only access keys that have a single word as the key and not a key that consists of a string of multiple words. Using square bracket notation, you can access the values as follows:

```
// Accessing values using square bracket notation
console.log(person['name']);       // "Alice"
console.log(person['age']);        // 30
console.log(person['isStudent']);  // false
```

Both notations achieve the same result, but square bracket notation is more flexible and allows you to access object properties with variable keys. For example:

```
let property = "age";
console.log(person[property]); // 30
```

Array:

Arrays represents an ordered collection of values, indexed by numbers (zero-based). Arrays can hold values of any data type, including other arrays. For example:

```
let fruits = ["apple", "banana", "cherry"];
```

You can access the elements of the array using index and starting from 0, as follows:

```
let firstFruit = fruits[0]; // "apple"
let secondFruit = fruits[1]; // "banana"
let thirdFruit = fruits[2]; // "cherry"
```

You can also modify the elements using index as follows:

```
fruits[1] = "grape"; // Modifying the second element
console.log(fruits); // ["apple", "grape", "cherry"]
```

Nesting Different Data Types:

Nesting different data types can be helpful when placing data structures or values of one data type inside another data structure of a different data type.

Nesting arrays in arrays:

```
const nestedArray = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
console.log(nestedArray);
// Output: [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

Nesting objects in objects:

```javascript
const nestedObject = {
    name: "John",
    address: {
        street: "123 Main St",
        city: "New York",
        zipCode: "10001"
    }
};
console.log(nestedObject);
/*
Output:
{
  name: 'John',
  address: { street: '123 Main St', city: 'New York', zipCode: '10001'
}
}
*/
```

Nesting objects in arrays:

```
const arrayOfObjects = [
    { name: "Alice", age: 25 },
    { name: "Bob", age: 30 },
    { name: "Charlie", age: 35 }
];
console.log(arrayOfObjects);
/*
Output:
[
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 35 }
]
*/
```

Nesting arrays in objects:

```
const objectWithArrays = {
    fruits: ["apple", "banana", "orange"],
    vegetables: ["carrot", "broccoli", "spinach"]
};
console.log(objectWithArrays);
/*
Output:
{
  fruits: [ 'apple', 'banana', 'orange' ],
  vegetables: [ 'carrot', 'broccoli', 'spinach' ]
}
*/
```