Event Listeners

```javascript
// JavaScript
const button = document.getElementById('myButton');

function handleClick() {
  alert('Button clicked!');
}

// Attach event listener
button.addEventListener('click', handleClick);

// Remove event listener
button.removeEventListener('click', handleClick);
```

1. Event listeners are a more flexible way to handle events, allowing multiple handlers for the same event and easier removal of handlers.
2. The addEventListener method is used to attach event listeners to elements.

Event Propagation (Bubbling)

```
// HTML:
// <div id="outerDiv">
//   <button id="innerButton">Click Me</button>
// </div>

// JavaScript
const outerDiv = document.getElementById('outerDiv');
const innerButton = document.getElementById('innerButton');

outerDiv.addEventListener('click', () => {
  console.log('Outer DIV clicked (bubbling).');
});

innerButton.addEventListener('click', () => {
  console.log('Inner Button clicked.');
});

// Clicking the button will log:
// "Inner Button clicked."
// "Outer DIV clicked (bubbling)."
```

1. Event propagation determines the order in which event handlers are executed when an event occurs on an element nested within other elements.
2. The event starts from the target element and bubbles up to the outer elements.
3. By default, events propagate in the bubbling phase.

Event Propagation (Stopping Propagation)

```
// JavaScript
innerButton.addEventListener('click', (event) => {
  console.log('Inner Button clicked.');
  event.stopPropagation(); // Stop the event from bubbling up
});

outerDiv.addEventListener('click', () => {
  console.log('Outer DIV clicked.');
});

// Clicking the button will log only:
// "Inner Button clicked."
```

You can stop the propagation of an event using the stopPropagation method.

Event Propagation (Capturing)

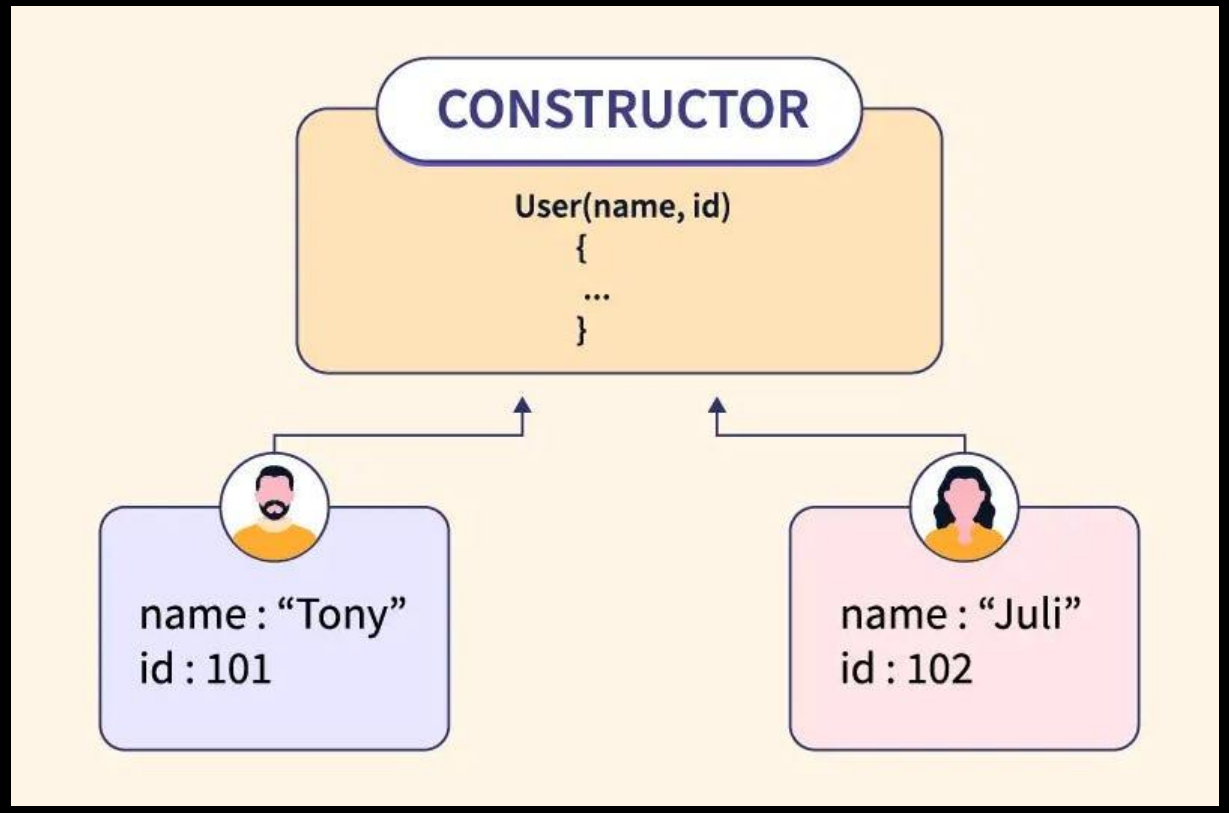
```javascript
// JavaScript
const outerDiv = document.getElementById('outerDiv');
const innerButton = document.getElementById('innerButton');

outerDiv.addEventListener(
  'click',
  () => {
    console.log('Outer DIV clicked (capturing).');
  },
  true // Enable capturing phase
);

innerButton.addEventListener('click', () => {
  console.log('Inner Button clicked.');
});

// Clicking the button will log:
// "Outer DIV clicked (capturing)."
// "Inner Button clicked."
```

1. The event starts from the outermost element and captures down to the target element.
2. Use the third parameter of addEventListener to specify capturing mode.
3. In this phase, the event starts from the window object and propagates down through the DOM tree to the target element.
4. Capturing phase is useful when you want to handle events at a higher level in the DOM hierarchy before they reach their target.

Constructor vs Object

Constructor is a blueprint; Objects are real values in memory.

Constructors



1. Constructors are special functions in JavaScript used to create and initialize objects.
2. They serve as blueprints for creating instances of specific types, making code organized and reusable.

Without Constructors

```
// User object with properties
const user1 = {
  name: "Prashant Jain",
  age: 32,
  email: "prashant@example.com",
  isAdmin: false,
};

const user2 = {
  name: "Shiv",
  age: 19,
  email: "shiv@example.com",
  isAdmin: false,
};

// Function to update user's admin status
function makeUserAdmin(user) {
  user.isAdmin = true;
}

makeUserAdmin(user1); // Make user an admin
```

1. Object literals are used to define objects, leading to code duplication and inefficiency.
2. Ensuring consistency in object structure requires manual validation, increasing the risk of errors.
3. Functions like makeUserAdmin are separate from objects, leading to less organized code.

Using Constructors

```
// User constructor function
function User(name, age, email, isAdmin) {
  this.name = name;
  this.age = age;
  this.email = email;
  this.isAdmin = isAdmin;

  // Method to update user's admin status
  this.makeUserAdmin = function() {
    this.isAdmin = true;
  };
}

// Create user1 object using the User constructor
const user1 = new User("Prashant Jain", 32,
  "prashant@example.com", false);

// Create user2 object using the User constructor
const user2 = new User("Shiv", 19,
  "shiv@example.com", false);

// Make user1 an admin
user1.makeUserAdmin();
```

1. By convention, constructor functions are named with an initial capital letter to distinguish them from regular functions.
2. new Keyword: When a function is called with new, it becomes a constructor that creates a new object instance. Inside the constructor, this refers to the new object.
3. Constructors automatically return the new object unless they explicitly return a different object.
4. Reusability: Constructors like Task provide a consistent structure for creating objects, reducing duplication and enhancing maintainability.
5. Consistency: Constructors ensure all objects have the same properties and methods, enforcing a common structure.
6. Encapsulation: Methods are encapsulated within objects, improving code organization and readability.

Constructor Prototype

```
// User constructor function
function User(name, age, email, isAdmin) {
  this.name = name;
  this.age = age;
  this.email = email;
  this.isAdmin = isAdmin;
}

// Method to update user's admin status using prototype
User.prototype.makeUserAdmin = function() {
  this.isAdmin = true;
};

// Create user1 object using the User constructor
const user1 = new User("Prashant Jain", 32,
  "prashant@example.com", false);

// Create user2 object using the User constructor
const user2 = new User("Shiv", 19,
  "shiv@example.com", false);

// Make user1 an admin
user1.makeUserAdmin();
```

Properties and methods can be added to a constructor's prototype to be shared across all instances created by that constructor.

Class (ES6 Convention)

```
// User constructor function
class User {
  constructor(name, age, email, isAdmin) {
    this.name = name;
    this.age = age;
    this.email = email;
    this.isAdmin = isAdmin;

    // Method to update user's admin status
    this.makeUserAdmin = function () {
      this.isAdmin = true;
    };
  }
}

// Create user1 object using the User constructor
const user1 = new User("Prashant Jain", 32,
  "prashant@example.com", false);

// Create user2 object using the User constructor
const user2 = new User("Shiv", 19,
  "shiv@example.com", false);

// Make user1 an admin
user1.makeUserAdmin();
```

- In ES6, the class keyword was introduced as syntactic sugar over constructor functions to define classes more succinctly.
- The instanceof operator can be used to check if an object is an instance of a particular constructor.

```
console.log(person instanceof Person); // Output: true
```

Class (ES6 Convention)

```
// Define a class
class Animal {
  // Constructor method
  constructor(name) {
    this.name = name;
  }

  // Method
  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

// Create an instance of the class
const animal = new Animal('Dog');
animal.speak(); // Output: "Dog makes a noise."
```

- Classes in JavaScript are a template for creating objects. They encapsulate data with code to work on that data.
- Introduced in ECMAScript 2015 (ES6), classes provide a clearer syntax to create and manage objects and handle inheritance.