1. Abstract vs Interface in JavaScript:

Abstract:

In JavaScript, abstract concepts are not directly supported as language features, unlike some other programming languages like Java or C#. However, developers often use the term "abstract" to describe objects or classes that serve as a blueprint but are not meant to be instantiated on their own. This is achieved through various techniques like using functions with the throw statement in their constructor to prevent instantiation or creating objects with methods that need to be implemented by subclasses.

Example:

```
function AbstractClass() {
    if (this.constructor === AbstractClass) {
        throw new Error("Cannot instantiate abstract class");
    }
    // Other abstract class initialization logic
}

AbstractClass.prototype.abstractMethod = function () {
    throw new Error("Abstract method must be implemented by subclass");
};

// Example usage
```

```
// var obj = new AbstractClass(); // This would throw an error
```

Interface:

JavaScript does not have native support for interfaces like some statically-typed languages, but developers often mimic interface behavior using a combination of conventions and documentation. An interface defines a contract for a set of methods that a class or object must implement.

Example:

```
function MyInterface() {}

MyInterface.prototype.method1 = function () {
    // Interface method 1
};

MyInterface.prototype.method2 = function () {
    // Interface method 2
};

// Example usage
function MyClass() {}

// Implementing the interface
MyClass.prototype.method1 = function () {
```

```
    // Implementation for method 1
};

MyClass.prototype.method2 = function () {
    // Implementation for method 2
};
```

2. Inheritance in Function Constructor:

In JavaScript, inheritance is often achieved through the prototype chain. When working with function constructors, you can set up inheritance by linking the prototypes of objects or classes.

```
// Parent constructor
function Parent(name) {
    this.name = name;
}

// Adding a method to the parent's prototype
Parent.prototype.greet = function () {
    console.log("Hello, " + this.name);
};

// Child constructor
function Child(name, age) {
    // Call the parent constructor with the current instance as 'this'
    Parent.call(this, name);
    this.age = age;
}

// Set up the prototype chain
Child.prototype = Object.create(Parent.prototype);
```

```
Child.prototype.constructor = Child;

// Adding a method specific to the child
Child.prototype.showAge = function () {
    console.log(this.name + " is " + this.age + " years old.");
};

// Example usage
var childObj = new Child("Alice", 25);
childObj.greet();   // Outputs: Hello, Alice
childObj.showAge(); // Outputs: Alice is 25 years old.
```

In the example above, the Child constructor inherits properties and methods from the Parent constructor using the Object.create() method to set up the prototype chain. The Child.prototype.constructor is also adjusted to point back to the Child constructor after setting up the prototype chain. This way, instances of Child can access both their own methods and properties and those of the Parent.