↳

Function Factories: Closures can be used to create functions that generate other functions with specific behavior. This is often used in scenarios like currying or creating functions with preset configurations.

Give an example code for this.

Certainly! Let's create an example of a function factory using closures to generate functions with specific behavior. We'll create a function that generates multiplier functions, which can multiply a given number by a preset factor.

```javascript
function createMultiplier(factor) {
    // This function is the factory function that generates multiplier functions
    return function(number) {
        // This inner function is the generated multiplier function
        return number * factor;
    };
}

// Create a multiplier function with a factor of 2
const double = createMultiplier(2);

// Create a multiplier function with a factor of 3
const triple = createMultiplier(3);

// Use the generated multiplier functions
console.log(double(5)); // Output: 10 (5 * 2)
```

```
console.log(triple(5)); // Output: 15 (5 * 3)
console.log(double(10)); // Output: 20 (10 * 2)
console.log(triple(10)); // Output: 30 (10 * 3)
```

In this example, `` `createMultiplier` `` is a function factory that takes a `` `factor` `` parameter and returns a new function. This returned function, referred to as a multiplier function, multiplies a given number by the preset `` `factor` ``. The `` `factor` `` value is captured by the closure, ensuring that each generated multiplier function retains its specific behavior.

By creating multiple multiplier functions with different factors, we can reuse the `` `createMultiplier` `` function to generate functions tailored to our needs. This demonstrates the power and versatility of function factories using closures in JavaScript.

I want you to replicate the same function factory example with some real world scenario/meaningfull code to understand and tell/compare them with normal code without closure and explain how the closure is helping for the code. Explain how it is different

Got it! Let's create a real-world scenario where closures and function factories can be beneficial. Suppose we are building a payment processing system where we need to calculate the total amount including taxes and discounts based on different configurations. We'll compare two implementations, one without closures and another with closures.

Without Closure:

```javascript
// Payment calculator without closure
function createPaymentCalculator(taxRate, discountRate) {
    return {
        calculateTotal: function(amount) {
            const taxAmount = amount * (taxRate / 100);
            const discountAmount = amount * (discountRate / 100);
            return amount + taxAmount - discountAmount;
```

```
        }
    };
}

const standardPaymentCalculator = createPaymentCalculator(10, 5);
console.log(standardPaymentCalculator.calculateTotal(100)); // Output: 105

// External code can directly modify taxRate and discountRate
standardPaymentCalculator.taxRate = 15;
standardPaymentCalculator.discountRate = 10;
console.log(standardPaymentCalculator.calculateTotal(100)); // Output: 110
```

In this example, the `createPaymentCalculator` function generates payment calculators based on the provided tax rate and discount rate. However, the tax and discount rates can be modified directly from outside the returned object, leading to potential issues such as unexpected calculation results.

With Closure:

```javascript
// Payment calculator with closure
function createPaymentCalculator(taxRate, discountRate) {
    return function(amount) {
        const taxAmount = amount * (taxRate / 100);
        const discountAmount = amount * (discountRate / 100);
        return amount + taxAmount - discountAmount;
    };
}

const standardPaymentCalculator = createPaymentCalculator(10, 5);
console.log(standardPaymentCalculator(100)); // Output: 105

// External code cannot directly modify taxRate and discountRate
standardPaymentCalculator.taxRate = 15;
standardPaymentCalculator.discountRate = 10;
console.log(standardPaymentCalculator(100)); // Output: 105
```

In this example, the `createPaymentCalculator` function returns an anonymous function that calculates the total amount based on the provided tax rate and discount rate. Since tax rate and discount rate are encapsulated within the closure, they cannot be directly modified from outside the returned function. This ensures that the calculation logic remains intact and prevents unintended changes to the tax and discount rates.

In summary, the closure-based implementation provides better encapsulation and data privacy compared to the non-closure-based implementation. It helps in maintaining the integrity of the calculation logic and prevents external code from inadvertently altering the configuration parameters.

↪

Without Closure:

javascript

Copy code

`// Payment calculator without closure function createPaymentCalculator(taxRate, discountRate) { return { calculateTotal: function(amount) { const taxAmount = amount * (taxRate / 100);

const discountAmount = amount * (discountRate / 100);   return amount + taxAmount - discountAmount;   }   }; } const standardPaymentCalculator = createPaymentCalculator(10, 5); console.log(standardPaymentCalculator.calculateTotal(100)); // Output: 105 // External code can directly modify taxRate and discountRate standardPaymentCalculator.taxRate = 15; standardPaymentCalculator.discountRate = 10; console.log(standardPaymentCalculator.calculateTotal(100)); // Output: 110`
I think this is also working with closure. Then what is the differnece?

You're correct. Both examples utilize closures. In both cases, the inner functions (e.g., `calculateTotal`) have access to variables (`taxRate` and `discountRate`) defined in their outer function (`createPaymentCalculator`). This access is facilitated by closures.

The difference lies in how the closure is utilized and the resulting behavior:

1. **Without Closure**: The closure is used to encapsulate the `calculateTotal` method within an object returned by `createPaymentCalculator`. However, the `taxRate` and `discountRate` are properties of this object, and they can be directly accessed and modified from outside. This breaks the encapsulation to some extent, as external code can manipulate these properties directly.

2. **With Closure**: The closure is used differently. Instead of encapsulating methods within an object, `createPaymentCalculator` directly returns an anonymous function that performs the payment calculation. The `taxRate` and `discountRate` are captured within the closure of this returned function, making them inaccessible from outside. This provides better encapsulation and data privacy, as external code cannot directly modify these variables.

In summary, while both examples use closures, the difference lies in how encapsulation and data privacy are achieved. The closure-based implementation provides stricter encapsulation by hiding variables from external access, while the non-closure-based implementation exposes these variables through an object interface.

But i am asking how the same use case are having drawbacks when we execute without using any closures. What is the difference between the applying this calculator with closure formation and the normal code which is not forming any closure. W

Why we the closure is helpfull in this case?

Use the code

Currying is a technique where a function that takes multiple arguments is transformed into a series of functions that take one argument each. It enables partial function application and enhances code flexibility.

Example: Let's create a curried function to calculate the total price of items with tax.

const calculateTotalPrice = (taxRate) => (price) => price + price * (taxRate / 100);

const calculateSalesTax = calculateTotalPrice(8); // 8% sales tax
const totalPrice = calculateSalesTax(100); // Price with tax
console.log(totalPrice); // 108

Let's break down the difference between using closures and not using closures in the context of a curried function for calculating total price with tax.

Without Closure:

```javascript
// Curried function without closure
const calculateTotalPrice = (taxRate, price) => price + price * (taxRate / 100);

const calculateSalesTax = (price) => calculateTotalPrice(8, price); // 8% sales tax
const totalPrice = calculateSalesTax(100); // Price with tax
console.log(totalPrice); // 108
```

In this example, we're calculating the total price with tax using a curried function. However, we're not utilizing closures. Instead, `calculateTotalPrice` and `calculateSalesTax` are separate functions, and `calculateSalesTax` directly calls `calculateTotalPrice` with the tax rate. There's no closure involved to encapsulate the `taxRate` variable.

Drawback: In this approach, the `taxRate` is not encapsulated within `calculateSalesTax`. This means that `calculateSalesTax` relies on an external variable (`taxRate`), making it less self-contained and potentially prone to errors if `taxRate` changes unexpectedly.

With Closure:

```javascript
// Curried function with closure
const calculateTotalPrice = (taxRate) => (price) => price + price * (taxRate / 100);

const calculateSalesTax = calculateTotalPrice(8); // 8% sales tax
const totalPrice = calculateSalesTax(100); // Price with tax
console.log(totalPrice); // 108
```

In this example, we're using closures to encapsulate the `taxRate` variable within the returned function from `calculateTotalPrice`. When `calculateSalesTax` is called, it's already bound to a specific `taxRate`, and it doesn't need to rely on external variables.

Advantage: Using closures ensures that each curried function (`calculateSalesTax` in this case) is self-contained and has its own `taxRate`. This improves code encapsulation, reduces dependencies on external variables, and makes the code more robust and predictable.

In summary, closures help in this case by providing better encapsulation and reducing dependencies, leading to cleaner and more maintainable code. They ensure that each curried function carries its own state (e.g., `taxRate`) without relying on external variables.

↳

Drawback: In this approach, the `taxRate` is not encapsulated within `calculateSalesTax`. This means that `calculateSalesTax` relies on an external variable (`taxRate`), making it less self-contained and potentially prone to errors if `taxRate` changes unexpectedly.

Can you explain in detail with an example showing how it is prone to errors?

Certainly! Let's delve deeper into how not encapsulating the `taxRate` variable within the `calculateSalesTax` function can make the code prone to errors.

Consider the scenario where we have a curried function for calculating total price with tax, but we don't use closures to encapsulate the `taxRate`. Instead, we rely on an external variable `taxRate`. Here's the code again for reference:

```javascript
// Curried function without closure
const calculateTotalPrice = (taxRate, price) => price + price * (taxRate / 100);

const calculateSalesTax = (price) => calculateTotalPrice(8, price); // 8% sales tax
const totalPrice = calculateSalesTax(100); // Price with tax
console.log(totalPrice); // 108
```

Now, let's explore potential issues with this approach:

1. **Dependency on External State**: The `calculateSalesTax` function relies on the `taxRate` variable being defined and accessible in the outer scope. If `taxRate` changes unexpectedly due to modifications elsewhere in the code, it can lead to incorrect calculations. For example, if another part of the code updates `taxRate` to `10` instead of `8`, `calculateSalesTax` will still use the old `taxRate` of `8`, leading to incorrect results.

2. **Global Namespace Pollution**: By not encapsulating `taxRate` within the function, we introduce the `taxRate` variable into the global namespace. This can lead to naming conflicts with other variables or functions in the global scope, potentially causing unintended behavior or errors.

3. **Code Maintenance Challenges**: Without encapsulation, it becomes harder to reason about the behavior of `calculateSalesTax` because it depends on external state. This can make the codebase more complex and harder to maintain, especially as it grows larger or when multiple developers are working on it.

4. **Testing Difficulty**: Testing becomes more challenging when functions rely on external state, as it's harder to isolate and control the variables affecting their behavior. This can result in less effective unit tests and potentially missing edge cases or bugs.

In summary, while the approach of not using closures may seem simpler at first, it introduces potential risks and complexities that can lead to errors and difficulties in maintaining the codebase over time. Encapsulating variables within closures helps mitigate these risks by making functions more self-contained, predictable, and easier to reason about.

↳

* **Dependency on External State**: The `calculateSalesTax` function relies on the `taxRate` variable being defined and accessible in the outer scope. If `taxRate` changes unexpectedly due to modifications elsewhere in the code, it can lead to incorrect calculations. For example, if another part of the code updates `taxRate` to `10` instead of `8`, `calculateSalesTax` will still use the old `taxRate` of `8`, leading to

incorrect results.

* **Global Namespace Pollution**: By not encapsulating `taxRate` within the function, we introduce the `taxRate` variable into the global namespace. This can lead to naming conflicts with other variables or functions in the global scope, potentially causing unintended behavior or errors.

* **Code Maintenance Challenges**: Without encapsulation, it becomes harder to reason about the behavior of `calculateSalesTax` because it depends on external state. This can make the codebase more complex and harder to maintain, especially as it grows larger or when multiple developers are working on it.

* **Testing Difficulty**: Testing becomes more challenging when functions rely on external state, as it's harder to isolate and control the variables affecting their behavior. This can result in less effective unit tests and potentially missing edge cases or bugs.
Can you write me a example code scenario of showing how this may happen.

Let's consider a scenario where the lack of encapsulation and dependency on external state can lead to errors. Suppose we have a retail application where the sales tax rate can vary depending on the user's location. We'll create a function to calculate the total price with tax, but we won't encapsulate the tax rate within the function. Instead, we'll rely on a global `taxRate` variable.

```javascript
let taxRate = 8; // Initial sales tax rate

// Function to calculate total price with tax
const calculateTotalPrice = (price) => price + price * (taxRate / 100);

// Function to calculate the final price for a product
const calculateFinalPrice = (productPrice) => {
    const totalPrice = calculateTotalPrice(productPrice);
    return totalPrice;
}

// Simulate a change in tax rate
setTimeout(() => {
    taxRate = 10; // Updated sales tax rate
    console.log("Tax rate updated to 10%.");
}, 2000);

// Calculate final price for a product
const productPrice = 100;
const finalPrice = calculateFinalPrice(productPrice);
console.log("Final price with tax:", finalPrice);
```

In this example, we have a global `taxRate` variable representing the sales tax rate. The `calculateTotalPrice` function calculates the total price with tax based on this `taxRate`. However, the `taxRate` variable is not encapsulated within the function, and it's susceptible to changes from external sources.

Now, let's simulate a scenario where the tax rate changes after a delay (e.g., due to a legislative update or a change in user location). We'll update the `taxRate` to `10%` after `2` seconds. Meanwhile, we calculate the final price for a product assuming the initial tax rate of `8%`.

When we run this code, the output may be unexpected:

```sql
Final price with tax: 108
Tax rate updated to 10%.
```

As you can see, the final price is calculated based on the initial tax rate of `8%`, even though the tax rate is updated to `10%` later. This discrepancy occurs because the `calculateTotalPrice` function relies on the global `taxRate`, and it doesn't account for changes to this variable.

This scenario highlights the dependency on external state and the lack of encapsulation, which can lead to incorrect calculations and unexpected behavior in real-world applications. Using closures to encapsulate variables like `taxRate` within functions can help mitigate these risks and ensure more reliable and predictable behavior.

↪

// Curried function without closure const calculateTotalPrice = (taxRate, price) => price + price * (taxRate / 100); const calculateSalesTax = (price) => calculateTotalPrice(8, price); // 8% sales tax const totalPrice = calculateSalesTax(100); // Price with tax console.log(totalPrice); // 108

Show/figure me out how the calculateSalesTax relies on external variable. Write me a sample code only using this code and show me what is external variable.

Certainly! In the provided example, `calculateSalesTax` relies on the external variable `taxRate`, which is defined outside the scope of the function. Let me demonstrate it by simplifying the code to focus only on `calculateSalesTax` and the external variable `taxRate`.

```javascript
let taxRate = 8; // External variable

// Function to calculate the total price with tax
const calculateSalesTax = (price) => price + price * (taxRate / 100);

// Simulate a change in tax rate after 2 seconds
setTimeout(() => {
    taxRate = 10; // Update external variable
    console.log("Tax rate updated to 10%.");
}, 2000);

// Calculate sales tax for a given price
const productPrice = 100;
const salesTax = calculateSalesTax(productPrice);
console.log("Sales tax for product price $100:", salesTax);
```

In this code, `calculateSalesTax` is defined as a function that calculates the total price with tax based on the external variable `taxRate`. This function does not explicitly receive `taxRate` as a parameter but relies on it being available in the outer scope.

The `setTimeout` function simulates a change in the tax rate after `2` seconds, updating the value of the external variable `taxRate` to `10%`.

When we call `calculateSalesTax` with a product price of `$100`, it calculates the total price with tax based on the initial value of `taxRate` (which is `8%`). However, after `2` seconds, the value of `taxRate` is updated to `10%`. Since `calculateSalesTax` relies on the external variable `taxRate`, it does not account for this change, leading to potential errors or inconsistencies in the calculation.

This example illustrates how `calculateSalesTax` depends on the external variable `taxRate` for its behavior, highlighting the risk of relying on external state without proper encapsulation.

// Curried function without closure
let taxRate = 8;
const calculateTotalPrice = (taxRate, price) => price + price * (taxRate / 100);

const calculateSalesTax = (price) => calculateTotalPrice(taxRate, price); // 8% sales tax
const totalPrice = calculateSalesTax(100); // Price with tax
console.log(totalPrice); // 108

can we write like this

Yes, absolutely! The code you provided is a valid way to implement a curried function without using closures, where the `taxRate` variable is defined externally. Let's break down your code:

```javascript
// Curried function without closure
let taxRate = 8; // External variable for tax rate

// Function to calculate the total price with tax
const calculateTotalPrice = (taxRate, price) => price + price * (taxRate / 100);

// Function to calculate the total price with tax, using the external tax rate
const calculateSalesTax = (price) => calculateTotalPrice(taxRate, price);

// Calculate total price with tax for a given price
const totalPrice = calculateSalesTax(100); // Price with tax
console.log(totalPrice); // Output: 108
```

In this code:

- `taxRate` is defined as an external variable representing the sales tax rate.
- `calculateTotalPrice` is a function that takes `taxRate` and `price` as parameters and calculates the total price with tax.
- `calculateSalesTax` is a function that takes `price` as a parameter and internally calls `calculateTotalPrice` with the external `taxRate`.
- `totalPrice` is calculated by calling `calculateSalesTax` with a price of `100`.

This approach works fine and produces the correct result, similar to using closures. However, it has the same drawback in terms of relying on external state, which can lead to potential issues if the external variables change unexpectedly.

Using closures to encapsulate variables like `taxRate` within functions can help mitigate these risks and ensure more reliable and predictable behavior. However, both approaches are valid, and the choice depends on the specific requirements and constraints of your application.

Explain what is difference between function with relies on external variable and function which is not relies on external variable in general?

Certainly! Let's explore the key differences between functions that rely on external variables and functions that do not rely on external variables:

Function Relies on External Variable:

1. **Dependency**: Functions that rely on external variables depend on values defined outside their scope. They cannot operate independently and require external state to be set correctly for proper execution.

2. **Encapsulation**: External variables are typically defined in a broader scope, such as the global scope or an enclosing function scope. This can lead to issues with encapsulation, as the function's behavior is intertwined with external state, making it less modular and harder to reason about.

3. **Global State**: Functions relying on external variables contribute to the global state of the application. This can lead to potential conflicts, naming collisions, and difficulties in tracking and managing state changes, especially in larger codebases or collaborative projects.

4. **Testing**: Testing functions that rely on external variables can be more challenging, as their behavior is influenced by external state. Setting up and controlling the necessary state for testing may require additional effort, and it can be harder to isolate and reproduce specific scenarios.

Function Does Not Rely on External Variable:

1. **Independence**: Functions that do not rely on external variables are self-contained and operate solely based on their inputs (parameters). They do not depend on values defined outside their scope and are thus more independent and predictable in their behavior.

2. **Encapsulation**: Since these functions do not rely on external variables, they are more encapsulated and modular. Their behavior is determined solely by their inputs, making them easier to understand, reuse, and maintain.

3. **Predictability**: Functions without external dependencies tend to be more predictable and easier to reason about. Their behavior is deterministic and consistent, regardless of changes in external state, leading to fewer surprises and potential bugs.

4. **Testing**: Testing functions without external dependencies is typically easier, as their behavior is isolated and deterministic. Tests can focus solely on providing different inputs and verifying the expected outputs, without needing to manage external state.

In summary, functions that do not rely on external variables tend to be more modular, predictable, and easier to test compared to functions that depend on external state. Encapsulation of variables within functions, often achieved through closures, can help reduce dependencies on external state and improve the overall design and maintainability of the code.

↳

const calculateSalesTax = (price) => calculateTotalPrice(8, price);
but here we are only relied/operate based on the input parameters only and not external variable. Explain what you mean by external variable here.

I see, let's clarify the concept of "external variable" in this context.

In the statement:

```javascript
const calculateSalesTax = (price) => calculateTotalPrice(8, price);
```

The function `calculateSalesTax` does not directly reference any variable declared outside its scope within its body. However, it does reference the `calculateTotalPrice` function, passing `8` as the tax rate argument.

In this case, `8` is a literal value and not a variable declared within the scope of `calculateSalesTax`. However, it's considered an external value because it's not passed as an argument to `calculateSalesTax` and is not declared within `calculateSalesTax` itself. Instead, it's hard-coded within the body of `calculateSalesTax`.

While `calculateSalesTax` doesn't directly rely on an external variable in the sense of a named variable declared outside its scope, it still relies on an external value (in this case, `8`) to perform its calculation. This value is external to the scope of `calculateSalesTax`, hence it's referred to as an external value or external constant.

In contrast, a function that doesn't rely on any external values would be one where all necessary values are passed as arguments to the function. For example:

```javascript
const calculateSalesTax = (taxRate, price) => calculateTotalPrice(taxRate, price);
```

In this case, `calculateSalesTax` relies only on the arguments `taxRate` and `price`, which are passed to it as parameters. There are no external values hard-coded within the function body.

// Curried function with closure
const calculateTotalPrice = (taxRate) => (price) => price + price * (taxRate / 100);

const calculateSalesTax = calculateTotalPrice(8); // 8% sales tax
const totalPrice = calculateSalesTax(100); // Price with tax
console.log(totalPrice); // 108

###

Explain how this code is not having external variable/hard coded value/literal value in the same context of last message.

In the provided code snippet, the function `calculateTotalPrice` is curried and uses closures to encapsulate the `taxRate` variable. Let's break down the code and see how it differs from relying on external variables:

```javascript
// Curried function with closure
const calculateTotalPrice = (taxRate) => (price) => price + price * (taxRate / 100);

const calculateSalesTax = calculateTotalPrice(8); // 8% sales tax
const totalPrice = calculateSalesTax(100); // Price with tax
console.log(totalPrice); // 108
```

1. **Encapsulation with Closure**: In the `calculateTotalPrice` function, the `taxRate` variable is encapsulated within the function's closure. When `calculateTotalPrice(8)` is called, it returns an inner function that has access to the `taxRate` variable through closure. This inner function is what actually calculates the total price with tax when invoked with a `price`.

2. **No External Variables/Literal Values**: Unlike the previous example where `8` was hard-coded directly into the function call, here `8` is a parameter passed to `calculateTotalPrice`. It's not hard-coded within the function body; instead, it's a value that can vary based on the context in which `calculateTotalPrice` is called.

3. **Dynamic Configuration**: By currying the `calculateTotalPrice` function, we create a reusable function `calculateSalesTax` that is configured with a specific tax rate (in this case, 8%). This makes the code more flexible and adaptable, as different tax rates can be used by simply calling `calculateTotalPrice` with a different value.

4. **No Direct Dependency on External State**: Unlike the previous example, where `calculateSalesTax` relied on a hard-coded value of `8`, here `calculateSalesTax` is a function that is configured with a specific tax rate (8%) during its creation. It doesn't directly depend on any external variables or hard-coded values within its scope.

In summary, while `calculateTotalPrice` and `calculateSalesTax` utilize closure to encapsulate the `taxRate` variable, they don't rely on external variables or hard-coded values within their scope. Instead, they create a self-contained and configurable mechanism for calculating total prices with tax.

↳

*   **Encapsulation with Closure**: In the `calculateTotalPrice` function, the `taxRate` variable is encapsulated within the function's closure. When `calculateTotalPrice(8)` is called, it returns an inner function that has access to the `taxRate` variable through closure. This inner function is what actually calculates the total price with tax when invoked with a `price`.

*   **No External Variables/Literal Values**: Unlike the previous example where `8` was hard-coded directly into the function call, here `8` is a parameter passed to `calculateTotalPrice`. It's not hard-coded within the function body; instead, it's a value that can vary based on the context in which `calculateTotalPrice` is called.

*   **Dynamic Configuration**: By currying the `calculateTotalPrice` function, we create a reusable

function `calculateSalesTax` that is configured with a specific tax rate (in this case, 8%). This makes the code more flexible and adaptable, as different tax rates can be used by simply calling `calculateTotalPrice` with a different value.

* **No Direct Dependency on External State**: Unlike the previous example, where `calculateSalesTax` relied on a hard-coded value of `8`, here `calculateSalesTax` is a function that is configured with a specific tax rate (8%) during its creation. It doesn't directly depend on any external variables or hard-coded values within its scope.
Explain exactly these four points what it will be in the opposite case.

↳

* **Encapsulation with Closure**: In the `calculateTotalPrice` function, the `taxRate` variable is encapsulated within the function's closure. When `calculateTotalPrice(8)` is called, it returns an inner function that has access to the `taxRate` variable through closure. This inner function is what actually calculates the total price with tax when invoked with a `price`.

* **No External Variables/Literal Values**: Unlike the previous example where `8` was hard-coded directly into the function call, here `8` is a parameter passed to `calculateTotalPrice`. It's not hard-coded within the function body; instead, it's a value that can vary based on the context in which `calculateTotalPrice` is called.

* **Dynamic Configuration**: By currying the `calculateTotalPrice` function, we create a reusable function `calculateSalesTax` that is configured with a specific tax rate (in this case, 8%). This makes the code more flexible and adaptable, as different tax rates can be used by simply calling `calculateTotalPrice` with a different value.

* **No Direct Dependency on External State**: Unlike the previous example, where `calculateSalesTax` relied on a hard-coded value of `8`, here `calculateSalesTax` is a function that is configured with a specific tax rate (8%) during its creation. It doesn't directly depend on any external variables or hard-coded values within its scope.

Explain exactly these four points what it will be in the opposite/other case.

Sure, let's contrast the characteristics of the code with closure (provided in the previous message) with those of code that relies on external variables:

1. **Encapsulation with Closure**:
    - *With Closure*: In the code using closure, the `` `taxRate` `` variable is encapsulated within the `` `calculateTotalPrice` `` function's closure. This means that `` `taxRate` `` is bound to the returned inner function and cannot be directly accessed or modified from outside.
    - *Without Closure*: In the absence of closure, if `` `taxRate` `` is declared as a global variable or defined in an outer scope, it's not encapsulated within the function. This exposes `` `taxRate` `` to potential modifications from other parts of the code, breaking encapsulation and potentially leading to unexpected behavior.
2. **No External Variables/Literal Values**:

- *With Closure*: The tax rate value (`8` in this case) is passed as a parameter to `calculateTotalPrice`. It's not hard-coded within the function body but rather provided as an argument during the function call.
- *Without Closure*: In contrast, without closure, the tax rate might be directly hard-coded or obtained from an external variable. If it's hard-coded within the function or obtained from an external source, changes to the tax rate require modifying the function's implementation or external state.

3. **Dynamic Configuration**:

   - *With Closure*: By currying the `calculateTotalPrice` function, we can create specialized functions (`calculateSalesTax`) configured with specific tax rates. Each invocation of `calculateTotalPrice` with a different tax rate creates a new function instance tailored to that rate.
   - *Without Closure*: Without currying or closure, the function's behavior might be static and not configurable. Changes to tax rates would require modifying the function's implementation directly or relying on external variables, making the code less flexible and adaptable.

4. **No Direct Dependency on External State**:

   - *With Closure*: Functions using closure and currying don't directly depend on external variables for their behavior. Instead, they encapsulate necessary state within their closures, leading to more predictable and self-contained functions.
   - *Without Closure*: Functions relying on external variables have a direct dependency on external state. Changes to external variables can impact the behavior of these functions, potentially leading to unintended consequences or bugs, especially in larger codebases or collaborative projects.

In summary, using closure and currying promotes encapsulation, flexibility, and predictability by allowing functions to be self-contained and configured with specific parameters. Conversely, functions that rely on external variables may suffer from issues related to encapsulation, reliance on global state, and reduced flexibility.