**Best Programming Practice Group Assignment**

**Group Members**

1. Muheto Hodal **29049**
2. Irakoze peace arlaine **23753**
3. Albert Dondo Sherman **25613**

**1. Factory Pattern**

**🔍 What it is:**  
The Factory pattern is like a factory in real life—it creates objects (products) for you based on some input or conditions. Instead of creating objects directly in your code, you ask the factory to make one for you.

**📦 Why use it?**  
It hides the object creation logic and makes your code more flexible. If you want to change how something is created, you only update the factory, not every place where it's used.

### 🧠Real-World Analogy: A Food Delivery Factory

**Imagine** you walk into a food delivery center (like a kitchen or cafeteria). You go to the counter and request a specific type of food, such as **Rice** or **Kawunga**. There's a staff member (our factory function) who listens to your request and **prepares the food package based on what you asked for**.

***// Factory function***

function foodFuctoryFunction(fullName, typeOfFood) { //***Function that will work behind***

if (typeOfFood === "rice") {

return {

typeOfFood: "Rice",

myFoodProduct: () =>

console.log(

"Hello again " +fullName +" This is Your Product Food: " +typeOfFood

),

};

} else if (typeOfFood === "kawunga") {

return {

typeOfFood: "Kawunga",

myFoodProduct: () =>

console.log(

"Hello again " + fullName + " This is Your Product Food: " +typeOfFood

),

};

} else {

return {

typeOfFood: "unKnown Food type",

myFoodProduct: () =>

console.log(

"Hello again " +fullName +" Your Product Food is " +typeOfFood +" and is unKnown Food type in this factory" ),

};

}

}

***Usage of the Function***

const newFoodProductRequest\_1 = foodFuctoryFunction("Muheto Hodal", "Umugati");

newFoodProductRequest\_1.myFoodProduct(); **//*OutPut: Hello again Muheto Hodal Your Product Food is Umugati and is unKnown Food type in this factory***

const newFoodProductRequest\_2 = foodFuctoryFunction("Umwiza Egeny", "rice");

newFoodProductRequest\_2.myFoodProduct(); /***/Hello again Umwiza Egeny This is Your Product Food: rice***

**Summary of the Analogy:**

🏭 The **factory function** acts like a food kitchen where **you place an order**, and based on what you request, **you get a specific meal box**. If the meal isn’t on the menu, the kitchen informs you politely. That’s exactly what the **Factory Pattern** does in code — **centralizes the creation logic** for various product types.

**2. Prototype Pattern**

**🔍 What it is:**  
The Prototype pattern is about creating new objects by copying an existing one instead of building a new one from scratch.

**📦 Why use it?**  
It’s useful when creating an object is expensive (like it takes a lot of time or memory), or when the object has many complex properties and it's easier to copy than rebuild.

**🧠 Real-World Analogy: Car Factory with Shared Instructions**

Imagine a **car manufacturing factory**. In this factory, all cars are manufactured from a common blueprint or instruction manual that provides the specifications for their design and operation. Instead of writing new driving instructions for every car, the factory just **reuses** the one manual.

const carPrototype = {

drive: function () {

console.log(`${this.brand} is driving 🚗`); },

};

function createCar(brand) {

const car = Object.create(carPrototype);

car.brand = brand;

return car;

}

***// Usage***

const car1 = createCar("Toyota");

car1.drive(); // Output: Toyota is driving 🚗

const car2 = createCar("Tesla");

car2.drive(); // Output: Tesla is driving 🚗

**3. Adapter Pattern**

**🔍 What it is:**  
The Adapter pattern allows two incompatible interfaces (or systems) to work together. It acts like a translator between them.

**📦 Why use it?**  
Sometimes, you want to reuse an existing class, but its methods don’t match the new system you’re building. The adapter helps bridge that gap.

**🧠 Real-world analogy:**  
You have a **European plug** but you're in a **US hotel**. You use a **power adapter** to make the European plug fit the US socket. That adapter doesn’t change the devices—it just makes them work together.

***// Old system***

class OldPrinter {

printOld(message) {

console.log("Old print: " + message);

}

}

***// Adapter***

class NewPrinterAdapter {

constructor() {

this.oldPrinter = new OldPrinter();

}

print(message) {

this.oldPrinter.printOld(message); // Convert new to old

}

}

***// Usage***

const printer = new NewPrinterAdapter();

printer.print("Hello from the adapter!"); // Output: Old print: Hello from the adapter!

**4. Proxy Pattern**

**🔍 What it is:**  
The Proxy pattern is like a middleman. It controls access to another object, usually for security, performance, or logging reasons.

**📦 Why use it?**  
You may want to add extra steps before allowing access to a real object. A proxy can check permissions, log actions, or delay loading a large object until it’s actually needed.

### 🏦 ****Real-World Analogy: Bank Account and Security Guard****

#### 👥 Imagine This:

There is a **bank account (realAccount)** containing money. But you can’t just walk in and start withdrawing money. There’s a **security guard (proxy)** at the door. Before letting anyone withdraw money, the guard checks **your role**.

Only the **admin** (like a bank manager) is allowed to access the money. If someone else (like a regular user or visitor) tries to withdraw, the guard stops them and says: **“Access denied! 🔐”**

const realAccount = {

balance: 1000,

withdraw(amount) {

if (amount <= this.balance) {

this.balance -= amount;

console.log(`Withdrawn $${amount}, Remaining: $${this.balance}`);

} else {

console.log("Insufficient balance.");

}

},

};

***// Proxy that checks permissions***

function createAccountProxy(role) {

return {

withdraw(amount) {

if (role !== "admin") {

console.log("Access denied! 🔐");

} else {

realAccount.withdraw(amount);

}

},

};

}

***// Usage***

const user = createAccountProxy("user");

user.withdraw(200); // Output: Access denied! 🔐

const admin = createAccountProxy("admin");

admin.withdraw(200); // Output: Withdrawn $200, Remaining: $800