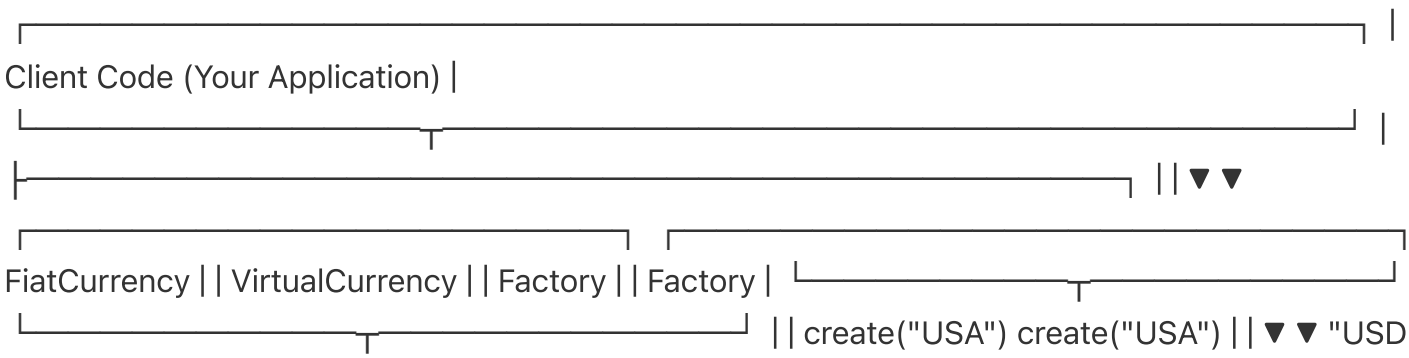
FACTORY METHOD PATTERN - COMPLETE LEARNING GUIDE **Author:** Aayush Tarwey **Date:** December 16, 2025 **Version:** 1.0 --- ## Table of Contents 1. [Quick Definition](#quick-definition) 2. [Real-World Analogy](#real-world-analogy) 3. [Pattern Components](#pattern-components) 4. [Detailed Explanation](#detailed-explanation) 5. [Code Walkthrough](#code-walkthrough) 6. [Comparison with Other Patterns](#comparison-with-other-patterns) 7. [Real-World Examples](#real-world-examples) 8. [Interview Q&A](#interview-qa) 9. [Key Takeaways](#key-takeaways) --- ## Quick Definition ### FACTORY METHOD PATTERN > A creational design pattern that defines an interface for creating objects, but lets subclasses decide which class to instantiate. **Key Point:** The Factory Method Pattern lets you create objects WITHOUT specifying their exact classes! **In Simple Words:** Instead of: ```python obj = ConcreteClass() # You create the object directly ``` Use Factory: ```python obj = factory.create() # Factory creates the object for you ``` --- ## Real-World Analogy ### 🏭 The Currency Exchange Booth **Without Factory Pattern:** ``` You need USD: 1. You go to the bank 2. You need to know how to create USD (complex process) 3. You manually create it 4. You receive USD ``` **With Factory Pattern:** ``` You need USD: 1. You go to the currency exchange booth 2. You say "I need USD" 3. The booth creates it for you (you don't know how) 4. You receive USD ``` **Same with our code:** ``` Without Factory: currency = "USD" # You create it manually With Factory: fiat_factory = FiatCurrencyFactory() currency = fiat_factory.create_country("USA") # Factory creates it ``` --- ## Pattern Components ### 1. **Product (Abstract Class)** ```python class Country: """Base class for all products""" pass ``` - Defines the interface for objects that factories will create - All concrete products inherit from this ### 2. **Concrete Products** ```python class USA(Country): pass class Spain(Country): pass ``` - Specific implementations of the product - Each represents a different type of object ### 3. **Creator/Factory (Abstract)** ```python class CountryFactory(ABC): @abstractmethod def create_country(self) -> Country: pass ``` - Defines the interface that ALL factories must follow - Ensures consistency across different factories - Uses `@abstractmethod` to force implementation ### 4. **Concrete Creators/Factories** ```python class FiatCurrencyFactory(CountryFactory): def create_country(self, country) -> str: # Implementation specific to fiat currencies pass class VirtualCurrencyFactory(CountryFactory): def create_country(self, country) -> str: # Implementation specific to virtual currencies pass ``` - Actual implementations that create specific types of products - Each factory handles one type of creation logic --- ## Detailed Explanation ### How the Factory Method Works #### **Step 1: Define What You Want to Create** ```python class Country: pass ``` This is the PRODUCT - what we want to create. #### **Step 2: Define the Factory Interface** ```python class CountryFactory(ABC): @abstractmethod def create_country(self) -> Country: pass ``` This FORCES all factories to have a `create_country()` method. #### **Step 3: Create Concrete Factories** ```python class FiatCurrencyFactory(CountryFactory): def create_country(self, country) -> str: if country == "USA": return "USD" elif country == "Spain": return "EUR" elif country == "Japan": return "JPY" ``` ```python class VirtualCurrencyFactory(CountryFactory): def create_country(self, country) -> str: if country == "USA": return "Bitcoin" elif country == "Spain": return "Ethereum" ``` **KEY INSIGHT:** Both

factories have the SAME METHOD but DIFFERENT IMPLEMENTATIONS! #### **Step 4: Use the Factories** ```python fiat = FiatCurrencyFactory() print(fiat.create_country("USA")) # Output: USD virtual = VirtualCurrencyFactory() print(virtual.create_country("USA")) # Output: Bitcoin ``` ### Visual Flow ```



``` --- ## Code Walkthrough ### File Structure ``` factory_method/ ├── src/ │ └── factory_method.py (Implementation with detailed comments) └── docs/ └── FACTORY_METHOD_GUIDE.md (This file) ``` ### Complete Code Example ```python from abc import ABC, abstractmethod # ============ PRODUCT ============ class Country: """Base product class""" pass # ======== CONCRETE PRODUCTS ======== class USA(Country): pass class Spain(Country): pass class Japan(Country): pass # ========= ABSTRACT FACTORY ========= class CountryFactory(ABC): """ Abstract factory that ALL concrete factories must follow This ensures every factory implements create_country() """ @abstractmethod def create_country(self, country) -> str: pass # ======= CONCRETE FACTORY #1 ======= class FiatCurrencyFactory(CountryFactory): """ Creates fiat currencies (USD, EUR, JPY, etc.) """ def create_country(self, country) -> str: if country == "USA": return "USD" elif country == "Spain": return "EUR" elif country == "Japan": return "JPY" else: return "Unknown Country" # ======= CONCRETE FACTORY #2 ======= class VirtualCurrencyFactory(CountryFactory): """ Creates virtual currencies (Bitcoin, Ethereum, etc.) """ def create_country(self, country) -> str: if country == "USA": return "Bitcoin" elif country == "Spain": return "Ethereum" elif country == "Japan": return "Dogecoin" else: return "Unknown Country" # ========== USAGE ========== if __name__ == "__main__": # Create factory instances fiat_factory = FiatCurrencyFactory() virtual_factory = VirtualCurrencyFactory() # Use factories to create objects print(fiat_factory.create_country("USA")) # USD print(virtual_factory.create_country("USA")) # Bitcoin ``` ### Code Explanation | Component | Purpose | Example | |-----------|---------|---------| | **Product** | Base class for all objects | `Country` | | **Concrete Products** | Specific types | `USA`, `Spain`, `Japan` | | **Factory Interface** | Rules all factories follow | `CountryFactory.create_country()` | | **Concrete Factories** | Creates specific products | `FiatCurrencyFactory`, `VirtualCurrencyFactory` | --- ## Comparison with Other Patterns ### Factory Method vs Simple Factory **Simple Factory:** ```python class CurrencyFactory: def create(self, type): if type == "fiat": return USD() elif type == "virtual": return Bitcoin() ``` - Single factory handles all types - Not easily extensible - All logic in one class **Factory Method (Better):** ```python class FiatCurrencyFactory(CountryFactory): def create_country(self, country): return USD()
```

Only handles fiat class VirtualCurrencyFactory(CountryFactory): def create_country(self, country): return Bitcoin() # Only handles virtual ``` - Different factories for different types - Easy to extend (add new factory) - Each factory has one responsibility ### Factory Method vs Abstract Factory **Factory Method:** - Creates ONE type of object - Example: `create_currency()` **Abstract Factory:** - Creates FAMILIES of related objects - Example: `create_currency()`, `create_payment_method()`, `create_wallet()` --- ## Real-World Examples ### Example 1: Database Connection Factories ```python class DatabaseConnection(ABC): @abstractmethod def connect(self): pass class MySQLFactory(DatabaseConnection): def connect(self): return "MySQL Connection established" class PostgreSQLFactory(DatabaseConnection): def connect(self): return "PostgreSQL Connection established" # Usage mysql = MySQLFactory() print(mysql.connect()) # MySQL Connection established postgres = PostgreSQLFactory() print(postgres.connect()) # PostgreSQL Connection established ``` ### Example 2: UI Button Factories (Cross-Platform) ```python class Button(ABC): @abstractmethod def render(self): pass class WindowsButtonFactory(Button): def render(self): return "Rendering Windows-style button" class MacButtonFactory(Button): def render(self): return "Rendering Mac-style button" # Usage windows_button = WindowsButtonFactory() mac_button = MacButtonFactory() windows_button.render() # Renders Windows-style mac_button.render() # Renders Mac-style ``` ### Example 3: Document Format Factories ```python class Document(ABC): @abstractmethod def save(self): pass class PDFDocumentFactory(Document): def save(self): return "Saving as PDF" class ExcelDocumentFactory(Document): def save(self): return "Saving as Excel" # Usage pdf = PDFDocumentFactory() excel = ExcelDocumentFactory() pdf.save() # Saving as PDF excel.save() # Saving as Excel ``` --- ## Interview Q&A ### Q1: What is the Factory Method Pattern? **A:** A creational design pattern that defines an interface for creating objects, but lets subclasses decide which class to instantiate. It helps you create objects without specifying their exact classes. ### Q2: Why would you use Factory Method instead of directly creating objects? **A:** - **Flexibility** - Easy to change which object is created - **Maintainability** - Object creation logic is in one place - **Loose Coupling** - Client doesn't know about concrete classes - **Extensibility** - Easy to add new object types ### Q3: What are the advantages of Factory Method? **A:** - ✅ Encapsulates object creation - ✅ Makes code more flexible - ✅ Follows Open/Closed Principle (open for extension, closed for modification) - ✅ Easy to unit test - ✅ Supports Single Responsibility Principle ### Q4: What are the disadvantages? **A:** - ❌ Can add unnecessary complexity for simple cases - ❌ Requires more classes - ❌ Can make code harder to follow initially ### Q5: How is Factory Method different from Abstract Factory? **A:** - **Factory Method:** Creates ONE product - **Abstract Factory:** Creates FAMILIES of related products ### Q6: What's the difference between Factory Method and Simple Factory? **A:** - **Simple Factory:** One factory handles all types (not extensible) - **Factory Method:** Different factories for different types (easily extensible) ### Q7: Can you use Factory Method with Singleton Pattern? **A:** Yes! You can have a Singleton factory that creates objects. This ensures only one factory exists. ### Q8: In the currency example, why use abstract methods? **A:** To force all concrete factories to implement the same interface, ensuring

consistency and making the code predictable. ### Q9: What real-world scenarios use Factory Method? **A:** - Database drivers (MySQL, PostgreSQL, MongoDB) - UI frameworks (buttons, textboxes for different OS) - Payment systems (Credit Card, PayPal, Crypto) - Document formats (PDF, Excel, Word) - File readers (CSV, JSON, XML) ### Q10: How would you add a new currency type to this code? **A:** ```python # Just add new concrete factory - no changes to existing code! class DigitalPaymentFactory(CountryFactory): def create_country(self, country) -> str: if country == "USA": return "PayPal" elif country == "Spain": return "Stripe" # etc... ``` ### Q11: Is Factory Method used in popular Python libraries? **A:** Yes! Examples: - Django ORM (database backend selection) - Pillow (image format handling) - SQLAlchemy (database connections) ### Q12: When should you NOT use Factory Method? **A:** - When you have only one product type - For very simple object creation - When it adds unnecessary complexity - When creation logic is trivial --- ## Key Takeaways ### ✅ What You Should Remember 1. **Factory Method creates objects** without specifying their exact classes 2. **Abstract Factory interface** ensures consistency across factories 3. **Concrete Factories implement** different creation logic 4. **Same method name, different results** - the power of polymorphism 5. **Easy to extend** - add new factory without changing existing code 6. **Encapsulates creation logic** - client doesn't need to know how objects are created 7. **Follows SOLID principles** - especially Open/Closed and Single Responsibility ### 📌 Common Mistakes to Avoid - ❌ Using Factory Method for simple single object creation - ❌ Not using abstract base classes - ❌ Making factories too complex - ❌ Not following the interface in concrete factories - ❌ Confusing with Simple Factory or Abstract Factory ### 💡 Interview Tips - **Explain with examples:** "Like a car factory - BMW factory creates BMWs, Toyota factory creates Toyotas" - **Know the differences:** Factory Method vs Simple Factory vs Abstract Factory - **Discuss trade-offs:** Flexibility vs complexity - **Be ready to extend:** How would you add a new currency type? - **Know real-world uses:** Database drivers, UI elements, payment systems - **Code it on whiteboard:** Be prepared to implement it --- ## Quick Reference ### When to Use - ✅ When you have multiple types of objects to create - ✅ When object creation logic is complex - ✅ When you want to decouple creation from usage - ✅ When you want to easily add new object types ### Pattern Structure ``` Client ↓ Abstract Factory (Interface) ↓ ├── Concrete Factory 1 ├── Concrete Factory 2 └── Concrete Factory 3 ↓ Product (created by factories) ``` ### Basic Template ```python from abc import ABC, abstractmethod # Product class Product(ABC): pass # Concrete Products class ProductA(Product): pass class ProductB(Product): pass # Factory class Factory(ABC): @abstractmethod def create(self): pass # Concrete Factories class FactoryA(Factory): def create(self): return ProductA() class FactoryB(Factory): def create(self): return ProductB() # Usage factory = FactoryA() product = factory.create() ``` --- **Happy Learning! 🚀** For more design patterns, return to the main README.md in the project root.