

OVERVIEW The Factory Method Pattern

Factory Method is a class-creational pattern: an abstract creator declares one factory method that returns a product interface, and each concrete creator subclass overrides it to decide which concrete product to instantiate. The creator's other methods use the product only through its interface. The lecture works through a computer store (StandardComputerStore makes StandardComputer, AdvancedComputerStore makes AdvancedComputer) in C++ and Java, lists the design steps and benefits, contrasts it with Abstract Factory, and practises it on document readers and notification senders.

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate; defer instantiation to subclasses.

Four roles

Creator (abstract, declares the factory method), ConcreteCreator (overrides it), Product (interface), ConcreteProduct.

One product vs. a family

Factory Method creates one product through one overridable method; Abstract Factory creates a family through a factory object with several methods.

Benefits

Code is separated from product-specific classes, so new products need only a new creator subclass (OCP); easier reuse, maintenance and testing.

HOW THE LECTURE IS ORGANIZED

Slides 1–4 Factory Method: intent and structure

Slides 5–13 Computer-store example: diagram, C++ and Java code

Slides 14–15 Design steps, benefits, summary

Slides 16–21 Exercise: document readers and parsers; Factory Method vs. Abstract Factory

Slides 22–24 Exercise: notification senders and channels

EXAM FOCUS

Not yet on a past paper, but it fits the Final's "which pattern? draw the class diagram" format (Final 252 Q6 did this for Builder and Singleton). Know the four roles, the code shape (abstract createX() overridden in subclasses) and the difference from Abstract Factory.

HOW TO READ THIS PAGE

Every original slide is shown unchanged. **Yellow marks** on a slide point to a numbered note: hover or tap a mark to jump to its note, and hover a note to see its marks. Questions tagged **Past exam** appeared on a real paper; **Prediction** questions are likely angles, not confirmed.



Software Design and Architecture

[Creational Design Patterns] – Chapter 06, L02

NOTES

- 1 Second creational-patterns lecture: **Factory Method**, the single-product cousin of Abstract Factory.

SLIDE 2 / 24 Lecture Outlines

Lecture Outlines

- **Factory Method Design Pattern**
 - ✓ Computer Store Example
- **Steps of Designing Factory Method Pattern**
- **Benefits of Factory Method**
- **What's next...**

NOTES

- 1 The pattern and the computer-store example.

2 The design steps.

3 The benefits.

SLIDE 3 / 24 Factory Method Design Pattern

Factory Method Design Pattern

➤ The Factory Method design pattern is a **class creational pattern** used to encapsulate and defer object instantiation to derived classes.

➤ The **intent** of the factory method is to:

- ✓ Define an interface for creating an object, but let subclasses decide which class to instantiate. **Factory method lets a class defer instantiation to subclasses.**

✓ Purpose

Defines **one method** for creating an object, but lets subclasses decide which class to instantiate.

◆ Key Idea

"Defer instantiation to subclasses."

◆ Creates:

→ **One product at a time**

◆ Structure:

1. **Creator** (abstract class or interface)
2. Concrete Creator
3. Product (interface)
4. Concrete Product

3

NOTES

- 1 Factory Method is a **class** creational pattern (it works through inheritance, so its relationships are fixed at design time) that encapsulates object creation and **defers instantiation to derived classes**.
- 2 **Intent:** define an interface for creating an object, but **let subclasses decide** which class to instantiate.
- 3 It creates **one product at a time** through **one** method. Key idea: "defer instantiation to subclasses".
- 4 **Structure:** (1) **Creator**, an abstract class or interface that declares the factory method; (2) **ConcreteCreator**, which overrides it; (3) **Product**, the interface every product implements; (4) **ConcreteProduct**.

Prediction

Q1

Which pattern lets subclasses decide which concrete class to instantiate through one overridable creation method?

- A. Factory Method
- B. Abstract Factory
- C. Singleton
- D. Builder

Answer: A. Factory Method

Prediction

Q2

State the intent of the Factory Method and name its four participants.

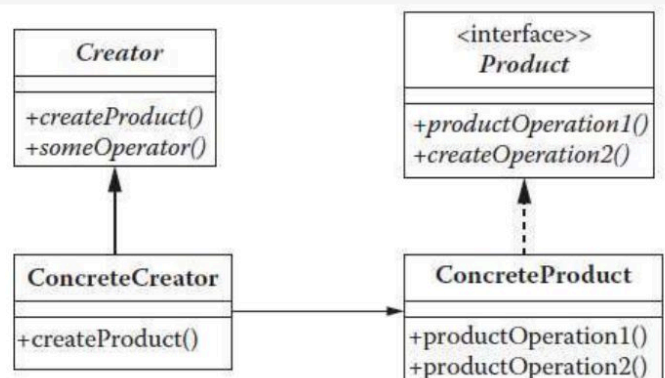
Answer: Define an interface for creating an object, but let subclasses decide which class to instantiate (defer instantiation to subclasses). Participants: Creator (declares the factory method), ConcreteCreator (overrides it), Product (interface), ConcreteProduct.

SLIDE 4 / 24 Factory Method Example

Factory Method Example

Structure

- The generic and applied structure of the factory method design pattern is presented below. As seen, the pattern requires both creational and product classes, similar to the abstract factory design pattern. However, unlike the abstract factory design pattern, **creator** classes in the factory method design pattern require **only one creational** interface method for creating products that share the same interface.
- Therefore, for each product in the system, the design incorporates **concrete product and concrete creator** classes.
- **Extra:**
- The Factory Method is simpler and is used when there is **one product type with multiple** implementations, while the Abstract Factory is used when there are multiple product types and multiple implementations of these product types.



NOTES

- 1 Like Abstract Factory it has creator and product classes, but the creator needs **only one** creational method, for products that share the same interface.
- 2 So for each product in the system there is one **concrete product** and one **concrete creator** class.

3 Rule of thumb: Factory Method for **one product type with several implementations**; Abstract Factory for **several product types**, each with several implementations.

4 Reading the generic diagram:

Creator

createProduct() in italics: abstract, the factory method. *someOperator()* is a normal method that calls *createProduct()* and uses the result.

ConcreteCreator

Inherits from Creator (solid line, hollow triangle) and implements *createProduct()*.

«interface» Product / ConcreteProduct

ConcreteProduct realises Product (dashed line). The arrow from ConcreteCreator to ConcreteProduct shows which product it creates.

Extra:

➤ Question (Scenario)

- A large electronics store wants to sell different types of computers. They currently have two categories:
 - ✓ **Standard Computers** (basic, affordable models)
 - ✓ **Advanced Computers** (high-end, gaming models)
- Each category of store can **only build its own type of computer**. For example:
 - ✓ The **Standard Store** can only build standard computers.
 - ✓ The **Advanced Store** can only build advanced computers.
- The store wants to provide a **common interface** so that when a customer asks for a computer, the store can return the correct computer type and also display its details (monitor, CPU, keyboard, cost).
- However, they want to design this in a way that makes it **easy to add more computer categories in the future** (e.g., "Workstation Computer Store" or "Budget Computer Store") **without changing existing code**.

1- Identify the Design Pattern

2- Write the Code

3- Draw the Class Diagram

NOTES

- 1 The scenario: an electronics store sells **standard** (affordable) and **advanced** (gaming) computers.
- 2 Each store type builds **only its own** kind: the Standard Store builds standard computers, the Advanced Store advanced ones.
- 3 Customers get a **common interface**: ask a store for a computer and display its details (monitor, CPU, keyboard, cost).
- 4 New categories (Workstation Store, Budget Store) must be addable **without changing existing code**.
- 5 Tasks: identify the pattern (Factory Method: one product type, a creator subclass per variant), write the code, draw the class diagram.

Factory Method Example

Extra:

1. ComputerStore (Abstract Class)

- Defines the abstract method `createComputer()` it is the factory method we use it for creating computer products. The return type is a pointer to a `Computer`.
- Declares `displayComputer()` for displaying computer information based on the 'type' (e.g., standard or advanced).

2. Derived Classes (StandardComputerStore, AdvancedComputerStore)

- `StandardComputerStore` and `AdvancedComputerStore` are concrete implementations of `ComputerStore`.
- Each class implements the `createComputer()` method to return a specific type of `Computer` (either Standard or Advanced).

3. Computer Interface

- The `Computer` interface defines the common methods (`displayMonitorInfo()`, `displayCpuInfo()`, etc.) that all computers (standard or advanced) must implement.

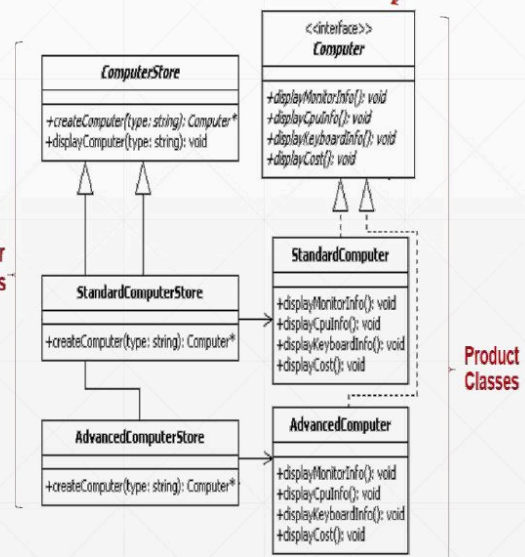
4. StandardComputer and AdvancedComputer

- Implement the `Computer` interface, each providing specific implementations for displaying information like monitor, CPU, keyboard, and cost.

Summary:

- The **Factory Method pattern** allows for the creation of different types of `Computer` objects (`StandardComputer` or `AdvancedComputer`) without changing the client code. The creation logic is encapsulated in the `ComputerStore` and its derived classes.

Creator
Classes

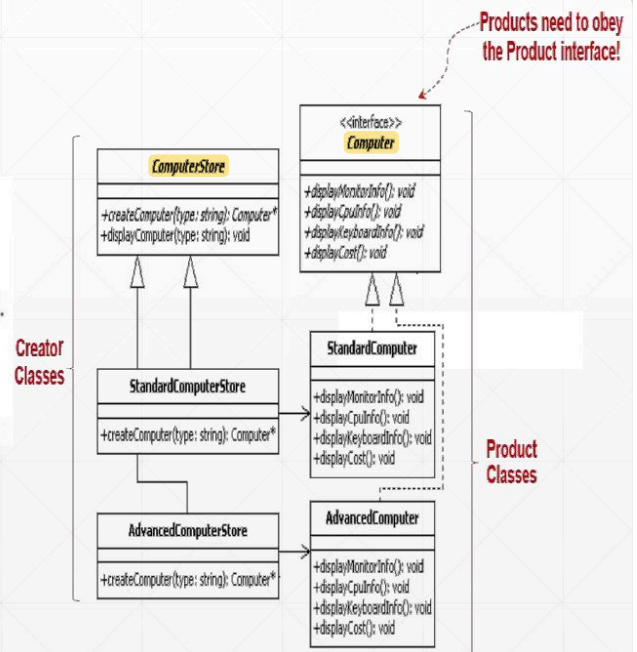


NOTES

- 1 **ComputerStore** (abstract creator): declares the abstract factory method `createComputer(type)`, returning a `Computer`, and `displayComputer(type)`, which displays the computer that `createComputer` produces.
- 2 **StandardComputerStore** and **AdvancedComputerStore** (concrete creators) each implement `createComputer()` to return their own kind of `Computer`.
- 3 **Computer** (product interface): `displayMonitorInfo()`, `displayCpuInfo()`, `displayKeyboardInfo()`, `displayCost()`. Note the italics in the diagram: abstract operations.
- 4 **StandardComputer** and **AdvancedComputer** (concrete products) implement those display methods.
- 5 Summary: different Computers are created without changing client code; the creation logic lives in `ComputerStore` and its subclasses.

Extra: Steps of Factory Method

1. Write `Computer` interface.
2. Write concrete classes: `StandardComputer`, `AdvancedComputer`
3. Write abstract `ComputerStore` with `createComputer()` + `displayComputer()`.
4. Write concrete stores: `StandardComputerStore`, `AdvancedComputerStore`.
5. Write `main` to test.



Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

7

NOTES

- 1 The writing order: (1) the `Computer` interface; (2) the concrete computers `StandardComputer`, `AdvancedComputer`; (3) the abstract `ComputerStore` with `createComputer()` and `displayComputer()`; (4) the concrete stores; (5) `main` to test. Products first, then creators, then the client: the same order you should use in an exam answer.

SLIDE 8 / 24 Factory Method Example

Factory Method Example

➤ Sample code for the `ComputerStore` Factory: **in C++**

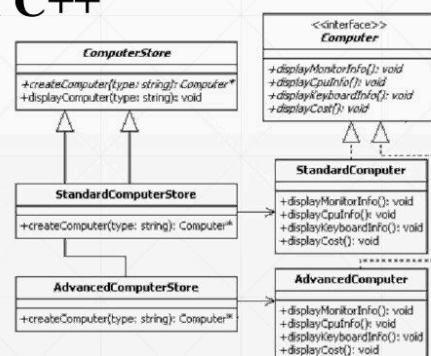
```
#include <string>

// Forward reference.
class Computer;

// The computer store creator class. This is an abstract class,
// therefore to instantiate computer stores, specific derived
// computer store classes are required.
class ComputerStore {

public:
    // The standard factory method for creating computer products.
    virtual Computer* createComputer(std::string type);

    // Method to display a computer's information.
    void displayComputer(std::string type);
};
```



Notes Extra:

- From code: Look for the **virtual** keyword to identify virtual methods "Abstract Methods".
- From diagrams: Look for methods written in **italics** (indicating they are abstract or virtual).

8

NOTES

1 The C++ creator class:

CODE

```
class Computer;
```

WHAT IT DOES

Forward declaration: ComputerStore only needs to know the type exists.

```
class ComputerStore {
```

The creator. It is abstract, so specific derived stores are needed.

```
    virtual Computer*  
    createComputer(std::string type);
```

The factory method: `virtual` lets derived classes override it (made pure virtual with `= 0` in a strictly abstract creator).

```
    void displayComputer(std::string  
type); };
```

A normal method that will use the created computer.

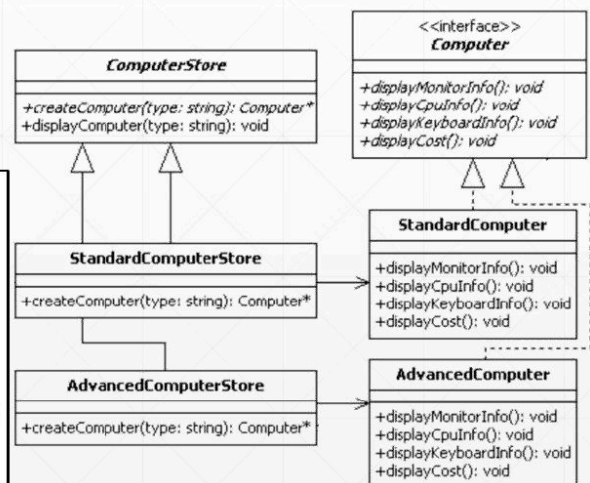
2 How to spot abstract methods: in code, look for the `virtual` keyword (C++) or `abstract` (Java); in UML diagrams, methods in *italics*.

SLIDE 9 / 24 Factory Method Example

Factory Method Example

➤ Sample code for the ComputerStore Factory:

```
#include "Computer.h"  
  
// Method to display a computer's information.  
void ComputerStore::displayComputer(string type) {  
  
    // Delegate the responsibility of creating a computer object to  
    // derived classes using the factory method.  
    Computer* computer = createComputer(type);  
  
    // Display the computer information, including its cost. This  
    // information varies according to the factory object used to create  
    // the computer.  
    computer->displayMonitorInfo();  
    computer->displayCpuInfo();  
    computer->displayKeyboardInfo();  
    computer->displayCost();  
  
    // Do more stuff with the computer object here.  
    // Clean up the pComputer and pFactory objects when done.  
}
```



1 displayComputer in C++:

CODE

```
void
ComputerStore::displayComputer(string
type) {
```

```
    Computer* computer =
createComputer(type);
```

```
    computer->displayMonitorInfo();
computer->displayCpuInfo();
```

```
    computer->displayKeyboardInfo();
computer->displayCost(); }
```

WHAT IT DOES

Defined once, in the abstract creator.

Delegates creation to the derived class through the factory method: this line is the whole pattern.

Uses the product only through the Computer interface.

The output differs by store, but this code never changes.

SLIDE 10 / 24 Factory Method Example

Factory Method Example

➤ Sample code for the ComputerStore Factory:

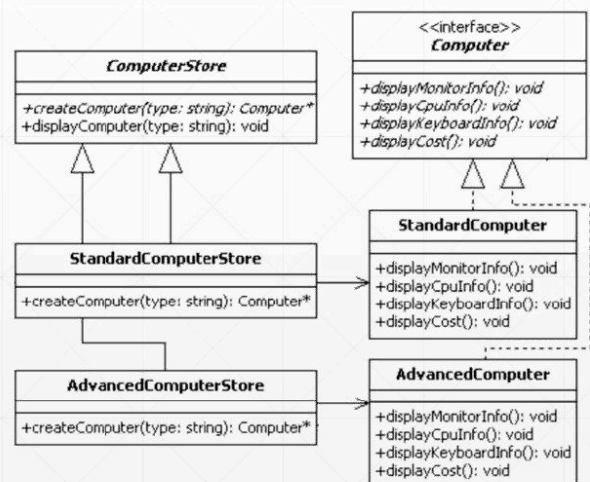
```
// Implement the factory method.
Computer* StandardComputerStore::createComputer(string type) {
    // Pointer to a computer object.
    Computer* computer;

    // Determine which computer needs to be created.
    if( type.compare("standard") == 0 {

        // Create the StandardComputer. Clients are responsible for
        // cleaning up the memory for the computer object. Internally,
        // StandardComputer uses StandardComputerPartsFactory to create
        // a standard computer.
        computer = new StandardComputer;
    }
    else {

        // Create and return a null computer.
        computer = new NullComputer;
    }

    // Return the newly created computer object. Clients are responsible
    // for cleaning up the computer object.
    return computer;
}
```



NOTES

1 A concrete creator in C++:

CODE

```
Computer*
StandardComputerStore::createComputer(
string type) {
```

```
    if (type.compare("standard") == 0)
        computer = new StandardComputer;
```

```
    else computer = new NullComputer;
```

```
    return computer; }
```

WHAT IT DOES

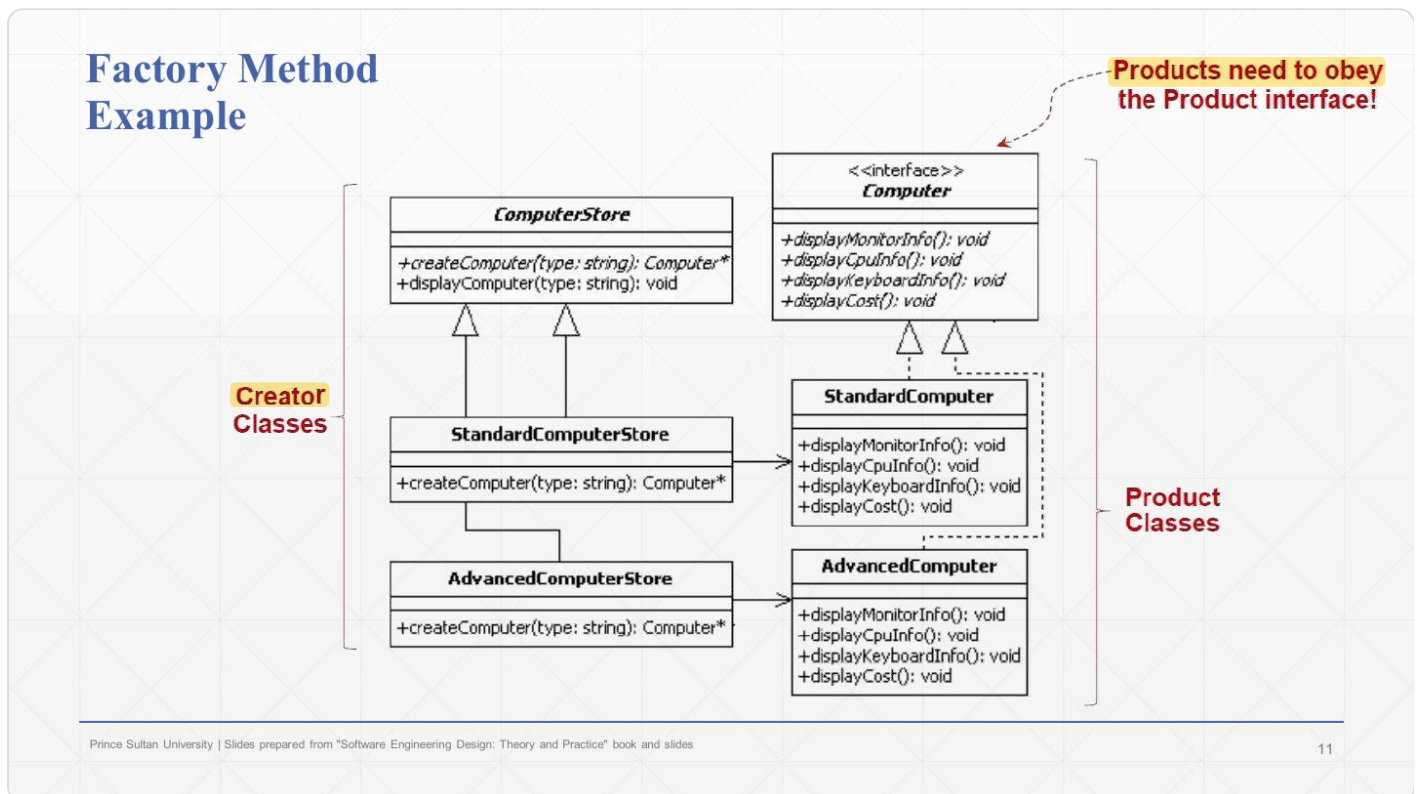
The standard store's override of the factory method.

It can only build standard computers.

Anything else returns a NullComputer: an object that does nothing, so callers never get a null pointer.

Clients are responsible for freeing the returned object (C++).

SLIDE 11 / 24 Factory Method Example: creator and product classes



NOTES

- The complete example diagram. **Creator classes:** abstract ComputerStore (createComputer(type): Computer, displayComputer(type)) with subclasses StandardComputerStore and AdvancedComputerStore.
- Product classes:** «interface» Computer, realised by StandardComputer and AdvancedComputer; the arrows from each store to its computer show which product each creator makes. Products must obey the product interface.

Steps for designing with the Factory Method

1. Identify and design the product interface (e.g., Computer)
2. Identify and design the concrete products that realize the interface from step 1 (e.g., StandardComputer, AdvancedComputer, etc.)
3. Design the factory interface (e.g., ComputerStore), which contains one abstract factory interface method for delegating product creation to derived classes.
4. Design one or more concrete factories for each product identified in step 2.
5. Associate each factory from step 4 with its respective product from step 2.

NOTES

- 1 **Step 1:** identify and design the product interface (Computer).
- 2 **Step 2:** design the concrete products that realise it (StandardComputer, AdvancedComputer).
- 3 **Step 3:** design the factory (creator) interface, ComputerStore, with **one abstract factory method** that delegates creation to derived classes.
- 4 **Step 4:** design one concrete factory (creator) per product from step 2.
- 5 **Step 5:** associate each factory with its product.

EXAM ANGLE

Prediction

Q1

Apply the Factory Method to a logistics app where RoadLogistics creates Trucks and SeaLogistics creates Ships. Name the classes and the factory method.

Answer: Product interface: Transport (deliver()). Concrete products: Truck, Ship. Abstract creator: Logistics with abstract createTransport(): Transport and planDelivery() that calls createTransport().deliver(). Concrete creators: RoadLogistics returns new Truck(); SeaLogistics returns new Ship().

```

abstract class ComputerStore {

    /** Factory Method – implemented by subclasses */
    public abstract Computer createComputer(String type);

    /** Helper to format all product info for the UI */
    public String displayComputer(String type) {
        Computer c = createComputer(type); } }

class StandardComputerStore extends ComputerStore {
    @Override
    public Computer createComputer(String type) {
        if ("standard".equalsIgnoreCase(type)) {
            return new StandardComputer();
        }
        // For anything else, return a NullComputer
        return new NullComputer("Standard store only builds 'standard' computers");
    }
}

class AdvancedComputerStore extends ComputerStore {
    @Override
    public Computer createComputer(String type) {
        if ("advanced".equalsIgnoreCase(type)) {
            return new AdvancedComputer();
        }
        return new NullComputer("Advanced store only builds 'advanced' computers");
    }
}

```

Extra

```

interface Computer {
    String displayMonitorInfo();
    String displayCpuInfo();
    String displayKeyboardInfo();
    String displayCost();
}

class StandardComputer implements Computer {
    @Override public String displayMonitorInfo() { return "Monitor: 24" 1080p"; }
    @Override public String displayCpuInfo() { return "CPU: Intel i5 / 8GB RAM"; }
    @Override public String displayKeyboardInfo() { return "Keyboard: Membrane (basic)"; }
    @Override public String displayCost() { return "Cost: $699"; }
}

class AdvancedComputer implements Computer {
    @Override public String displayMonitorInfo() { return "Monitor: 27" 1440p, 144Hz"; }
    @Override public String displayCpuInfo() { return "CPU: Intel i7 / 32GB RAM + RTX 4070"; }
    @Override public String displayKeyboardInfo() { return "Keyboard: Mechanical RGB"; }
    @Override public String displayCost() { return "Cost: $1999"; }
}

public static void main(String[] args) {
    // Create stores
    ComputerStore standardStore = new StandardComputerStore();
    ComputerStore advancedStore = new AdvancedComputerStore();

    System.out.println("=== Standard Store, Standard Computer ===");
    standardStore.displayComputer("standard");

    System.out.println("\n=== Advanced Store, Advanced Computer ===");
    advancedStore.displayComputer("advanced");

    System.out.println("\n=== Mismatched Example: Advanced Store asked for 'standard' ===");
    advancedStore.displayComputer("standard");
}

```

NOTES

- 1 The same example in Java:

CODE

```

abstract class ComputerStore { public
abstract Computer
createComputer(String type);

```

```

    public String
displayComputer(String type) {
Computer c = createComputer(type); ... }

```

```

class StandardComputerStore extends
ComputerStore { @Override public
Computer createComputer(String type) {

```

```

    if
("standard".equalsIgnoreCase(type))
return new StandardComputer();

```

```

        return new NullComputer("Standard
store only builds 'standard'
computers"); } }

```

```

class AdvancedComputerStore extends
ComputerStore { ... new
AdvancedComputer() ... }

```

WHAT IT DOES

The factory method, implemented by subclasses.

A helper that formats the product info, using only Computer.

The standard creator...

...builds a StandardComputer...

...or a NullComputer with a message.

The advanced creator, same shape.

2 Products and client:

CODE	WHAT IT DOES
<pre>interface Computer { String displayMonitorInfo(); String displayCpuInfo(); String displayKeyboardInfo(); String displayCost(); }</pre>	The product interface.
<pre>class StandardComputer implements Computer { ... "Monitor: 24\" 1080p" ... "Cost: \$699" }</pre>	Concrete products return their own details.
<pre>ComputerStore standardStore = new StandardComputerStore(); standardStore.displayComputer("standar d");</pre>	The client works through ComputerStore.
<pre>advancedStore.displayComputer("standar d");</pre>	The mismatch example: the advanced store asked for a standard computer returns the NullComputer message.

SLIDE 14 / 24 Benefits of the Factory Method Pattern

Benefits of the Factory Method Pattern

- Separates code from product-specific classes; therefore, the same code can work with various existing or newly created product classes.
- The development becomes efficient, since different developers can work on the different parts of the project at the same time.
- Easier to reuse specific parts of the code
- Easier to maintain specific parts of the code.
- Improve Testability.

NOTES

- 1 It **separates code from product-specific classes**, so the same code works with existing and newly created products. New product = new creator subclass, no edits elsewhere (OCP).
- 2 Development is more **efficient**: different developers can work on different creators and products at the same time.
- 3 Specific parts are easier to **reuse** and **maintain**.

- 4 **Testability** improves: a test can subclass the creator to return a fake product.

EXAM ANGLE

Prediction

Q1

True or false: the Factory Method helps follow the Open-Closed Principle when new product types are added.

Answer: True — A new product needs a new concrete product and a new creator subclass; existing creators and client code are not modified.

SLIDE 15 / 24 Summary

Summary

- In this session, we continued the discussion on creational design patterns, including:
 - ✓ Factory Method
- In the next sessions, we will finalize the presentation on creational design patterns. Specifically, we will cover:
 - ✓ Builder
 - ✓ Singleton

NOTES

- 1 This lecture: **Factory Method**.
- 2 Next: **Builder** and **Singleton**, which complete the creational patterns (both were on Final 252).

Extra: Question(Scenario)

- A **Document Reader** framework must handle different file types (PDF, Word, Excel). So, this class is an abstract “**Document Reader**” base class that implements the overall workflow for reading a file. This base class declares a **single overridable method** whose job is to **return the parser** to use.
- There is:
 - **One common Parser Interface** used by all concrete parsers.
 - **Each concrete reader subclass** (e.g., PdfReader, WordReader, ExcelReader) overrides that single method to **provide the appropriate parser**.
 - To support a new type (e.g., PowerPoint, Text), you **add a new reader subclass** that returns the new parser—**no changes to existing client code** are needed.
 - **Which design pattern is most appropriate here?**
 - **Draw a UML diagram for your design?**
 - **Write a simple Java code example showing how the Factory Method?**

16

NOTES

- 1 Scenario: a document-reader framework handles PDF, Word and Excel files. An abstract **DocumentReader** base class implements the overall reading workflow.
- 2 It declares a **single overridable method** whose job is to return the parser to use: that is the factory method.
- 3 **One common Parser interface** is used by all concrete parsers (the product).
- 4 Each concrete reader (PdfReader, WordReader, ExcelReader) overrides that method to return the right parser; a new type (PowerPoint, Text) means a new reader subclass, with no client changes.
- 5 Answer: **Factory Method**. The next slides give the UML and code.

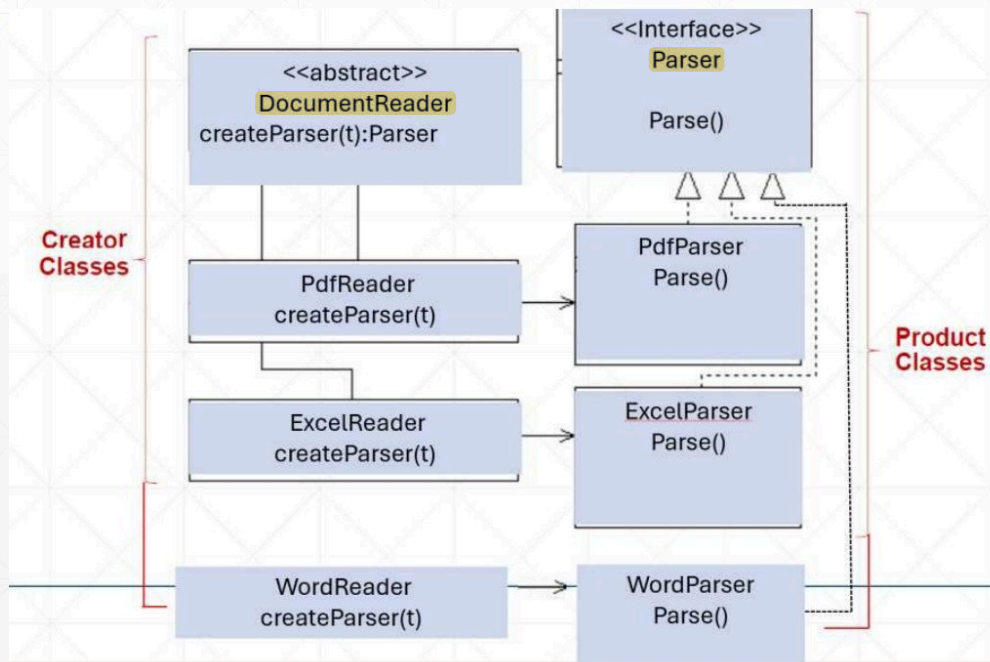
EXAM ANGLE

Prediction

Q1

A DocumentReader framework must create the right Parser for PDF, Word or Excel files through one overridable method. Which pattern, and what are the classes?

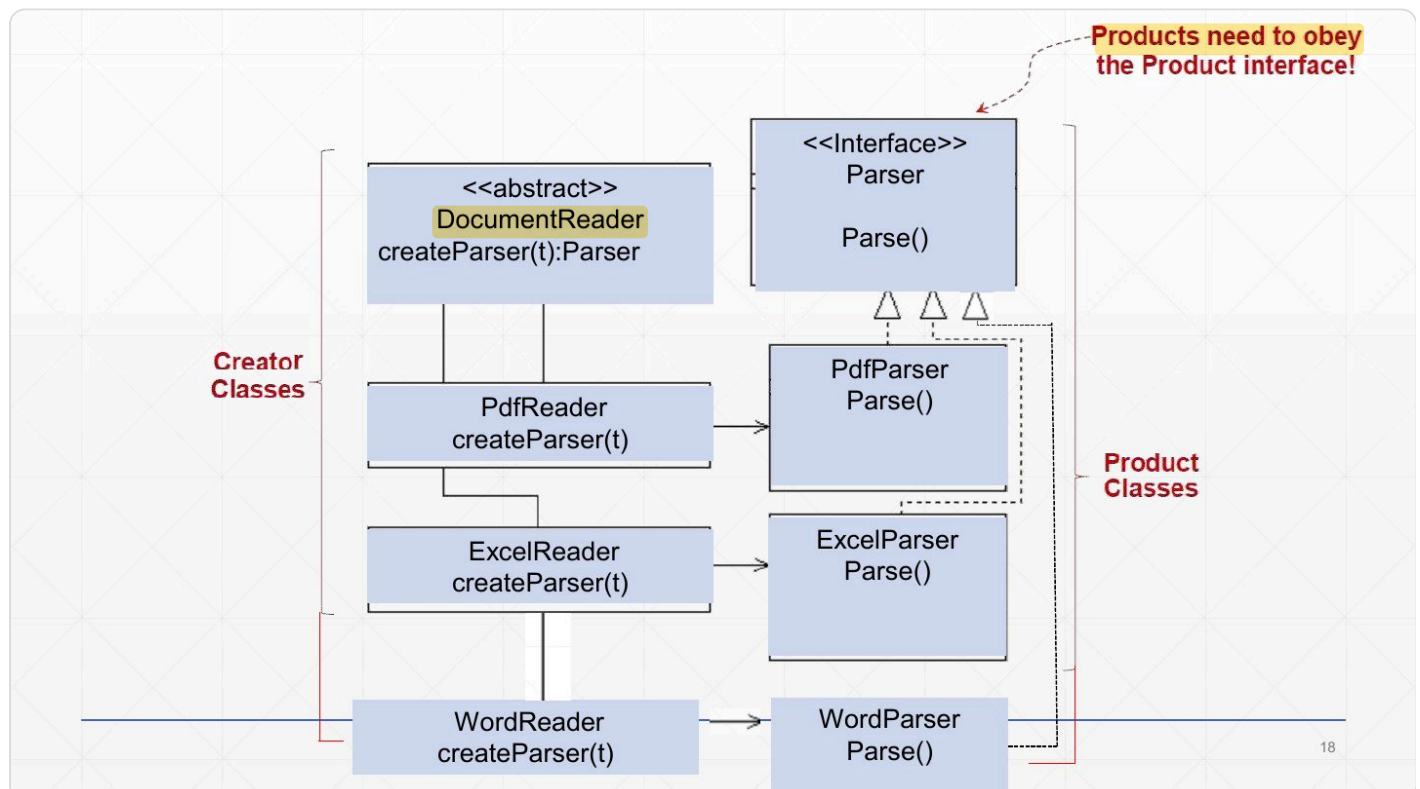
Answer: Factory Method. Parser is the product interface (parse()); PdfParser, WordParser, ExcelParser implement it. DocumentReader is the abstract creator with abstract createParser(): Parser and a read() workflow that calls createParser().parse(). PdfReader, WordReader and ExcelReader extend DocumentReader and return their parser.

Extra:

17

NOTES

- 1 The solution diagram: «abstract» **DocumentReader** with `createParser(t): Parser`; **PdfReader**, **ExcelReader** and **WordReader** extend it (creator classes).
- 2 «Interface» **Parser** with `Parse()`, realised by **PdfParser**, **ExcelParser** and **WordParser** (product classes). Each reader points to the parser it creates.



18

NOTES

- 1 The same diagram labelled with the pattern's roles: **creator classes** on the left (DocumentReader and its readers), **product classes** on the right (Parser and its parsers). Products must obey the Parser interface; each concrete reader is paired with exactly one concrete parser.

SLIDE 19 / 24 Extra: General order for writing classes

Extra:

- ◆ **General Order for Writing Classes**

1. **Product (Interface or Abstract Class)**
 - Define the common behavior that all concrete products will share.
 - Example: `Parser` with a method `parse()`.
2. **Concrete Products**
 - Implement the product interface for each variant.
 - Example: `PdfParser`, `WordParser`, `ExcelParser`.
3. **Creator (Abstract Class)**
 - Declare the **factory method** that will return the product.
 - Example: `DocumentReader` with `createParser()`.
4. **Concrete Creators**
 - Implement the factory method to return the correct product.
 - Example:
 - `PdfReader` → returns `PdfParser`
 - `WordReader` → returns `WordParser`
 - `ExcelReader` → returns `ExcelParser`
5. **Client**
 - Write the `main` method (or test class).
 - Use the **Creator** type (not concrete products directly) to get objects and run their behavior.

Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

19

NOTES

- 1 **1. Product:** the common behaviour (`Parser` with `parse()`).
- 2 **2. Concrete products:** one per variant (`PdfParser`, `WordParser`, `ExcelParser`).
- 3 **3. Creator:** the abstract class that declares the factory method (`DocumentReader` with `createParser()`).
- 4 **4. Concrete creators:** implement the factory method to return the right product (`PdfReader` → `PdfParser`...).
- 5 **5. Client:** `main` uses the **creator** type, never the concrete products directly.

```

// 1) ----- Creator (Abstract Class) -----
abstract class DocumentReader {
    public abstract Parser createParser(); // Factory Method
}

// 2) ----- Concrete Creators (Subclasses) -----
class PdfReader extends DocumentReader {
    public Parser createParser() {
        return new PdfParser();
    }
}

class WordReader extends DocumentReader {
    public Parser createParser() {
        return new WordParser();
    }
}

class ExcelReader extends DocumentReader {
    public Parser createParser() {
        return new ExcelParser();
    }
}

// 3) ----- Product (Interface) -----
interface Parser {
    void parse();
}

// 4) ----- Concrete Products (Implement Interface) -----
class PdfParser implements Parser {
    public void parse() {
        System.out.println("Parsing PDF...");
    }
}

class WordParser implements Parser {
    public void parse() {
        System.out.println("Parsing Word...");
    }
}

class ExcelParser implements Parser {
    public void parse() {
        System.out.println("Parsing Excel...");
    }
}

// ----- Client -----
public class FactoryMethodExample {
    public static void main(String[] args) {
        DocumentReader reader;

        reader = new PdfReader();
        reader.createParser().parse();

        reader = new WordReader();
        reader.createParser().parse();

        reader = new ExcelReader();
        reader.createParser().parse();
    }
}

```

NOTES**1** The Java solution:**CODE**

```
abstract class DocumentReader {
    public abstract Parser createParser(); //
    Factory Method
}
```

```
class PdfReader extends DocumentReader {
    public Parser createParser() {
        return new PdfParser();
    }
}
```

```
interface Parser {
    void parse();
}
```

```
class PdfParser implements Parser {
    public void parse() {
        System.out.println("Parsing PDF...");
    }
}
```

```
DocumentReader reader;
reader = new PdfReader();
reader.createParser().parse();
```

WHAT IT DOES

1) The creator.

2) Concrete creators, one per format (WordReader, ExcelReader are the same).

3) The product.

4) Concrete products.

5) The client holds a DocumentReader; switching to new WordReader() changes the output to "Parsing Word..." without other changes.

Extra:

◆ Differences Between Factory Method and Abstract Factory

Aspect	Factory Method	Abstract Factory
Purpose	Defines a single factory method to create one product at a time.	Provides an interface for creating families of related products .
Factory	One factory method (often parameterized). Subclasses decide which product to return.	A factory interface with multiple factory methods (one per product type).
Products	Usually deals with one hierarchy of products (e.g., <code>Parser</code> → <code>PdfParser</code> , <code>WordParser</code>).	Deals with multiple product hierarchies that are related (e.g., <code>Button</code> , <code>Checkbox</code> families for Windows, Mac).
Flexibility	Easier, more lightweight — useful when only one kind of product is needed.	More complex — ensures consistency among related products across a family.
Client dependency	Client calls a single method (e.g., <code>createProduct(type)</code> or <code>createParser()</code>).	Client calls multiple factory methods (e.g., <code>createButton()</code> , <code>createCheckbox()</code>), but all products are guaranteed to belong to the same family.
Example	A <code>DocumentReader</code> creates one parser (PDF, Word, or Excel).	A <code>GUIFactory</code> creates a set of widgets (Button + Checkbox) for one platform (Windows, Mac, Linux).

Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

21

NOTES

- 1 Purpose:** Factory Method defines a single factory method to create **one product** at a time; Abstract Factory provides an interface for creating **families** of related products.
- 2 Factory and products:** one (often parameterised) method whose subclasses decide the product, with one product hierarchy (`Parser` → `PdfParser`, `WordParser`); versus a factory interface with one method per product type and several related hierarchies (`Button`, `Checkbox` for Windows and Mac).
- 3 Flexibility and client:** Factory Method is lighter, used when only one kind of product is needed, and the client calls a single method (`createParser()`); Abstract Factory is more complex but guarantees consistency across a family, and the client calls several methods (`createButton()`, `createCheckbox()`). Examples: `DocumentReader` creates one parser; `GUIFactory` creates a set of widgets.

EXAM ANGLE

Prediction

Q1

Give two differences between the Factory Method and the Abstract Factory patterns.

Answer: Factory Method creates one product through one overridable method in a creator subclass (inheritance, class scope); Abstract Factory creates a family of related products through a factory object with one method per product type (composition, object scope). Factory Method has one product hierarchy; Abstract Factory has several related hierarchies and guarantees the family is consistent.

Extra: Question:

A **Notification Sender framework** must send notifications through different channels (**Email**, **SMS**, **Push Notification**).

The framework has an abstract base class **NotificationSender** that implements the overall workflow of sending a notification (validate message → create channel sender → send).

The base class declares **one overridable method** whose job is to return the **Channel** object to use.

There is:

- One common interface **Channel** used by all concrete channels.
- Each concrete sender subclass (e.g., **EmailSender**, **SmsSender**, **PushSender**) overrides that single method to provide the appropriate channel.
- To support a new channel (e.g., WhatsApp), you add a new sender subclass and a new channel class—no changes to existing client code.

Questions:

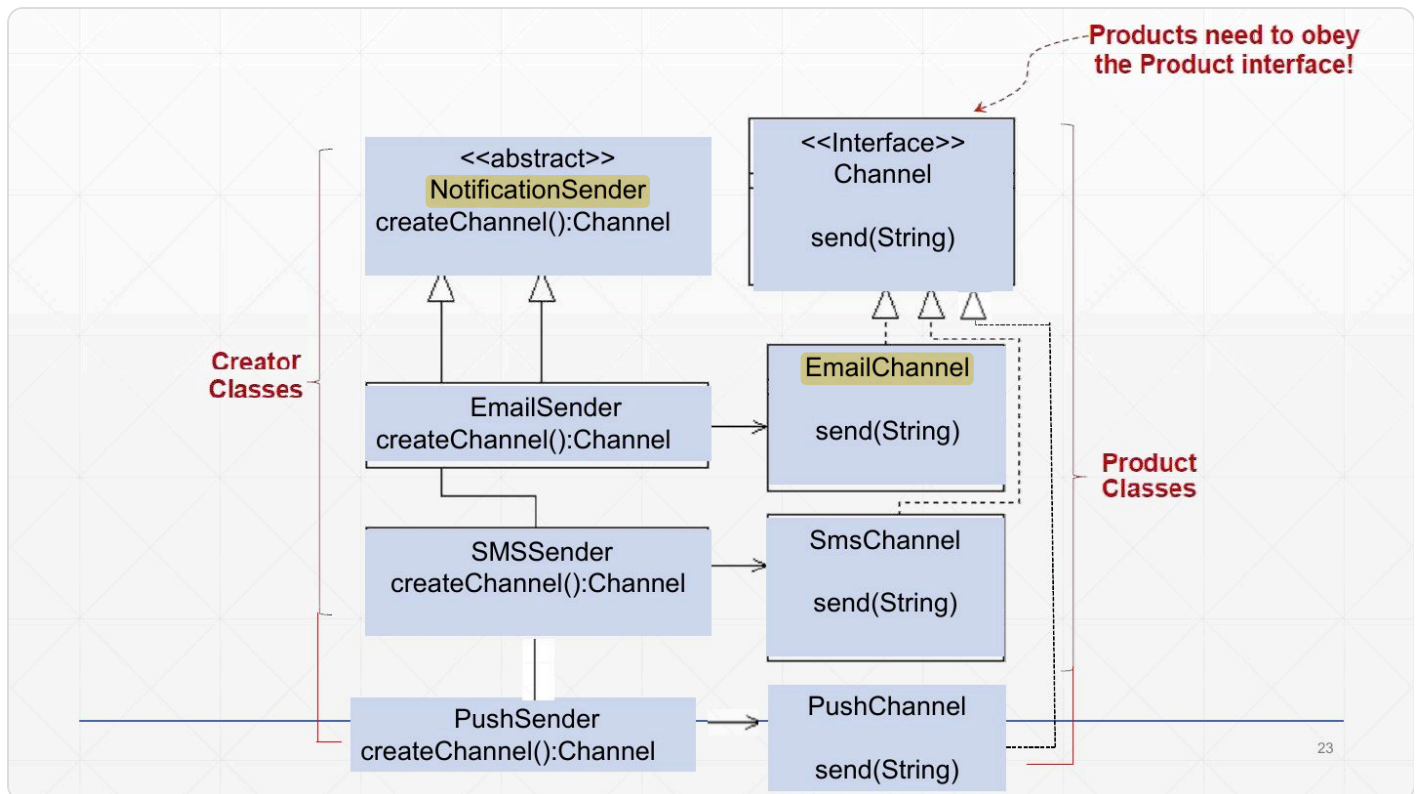
1. Which design pattern is most appropriate here?
2. Draw a UML diagram for your design.
3. Write a simple Java code example showing how the Factory Method works.

22

NOTES

- 1 A second exercise: a notification-sender framework sends messages through Email, SMS or Push. The abstract base class **NotificationSender** implements the workflow (validate message → create channel → send).
- 2 It declares **one overridable method** that returns the **Channel** to use; one common **Channel** interface; each sender subclass (**EmailSender**, **SmsSender**, **PushSender**) overrides that method; a new channel (WhatsApp) means a new sender and channel class, no client changes.
- 3 Questions: which pattern (Factory Method), draw the UML, write simple Java. Solutions on slides 23–24.

SLIDE 23 / 24 Notification Sender class diagram



NOTES

- 1 Creator classes: «abstract» **NotificationSender** with `createChannel(): Channel`; `EmailSender`, `SMSSender` and `PushSender` extend it.
- 2 Product classes: «Interface» **Channel** with `send(String)`, realised by `EmailChannel`, `SmsChannel` and `PushChannel`. Each sender creates its matching channel.

SLIDE 24 / 24 Extra: Notification Sender code

1 Product Interface

```
java
interface Channel {
    void send(String msg);
}
```

2 Concrete Products

```
java
class EmailChannel implements Channel {
    public void send(String msg) {
        System.out.println("Email: " + msg);
    }
}

class SmsChannel implements Channel {
    public void send(String msg) {
        System.out.println("SMS: " + msg);
    }
}

class PushChannel implements Channel {
    public void send(String msg) {
        System.out.println("Push: " + msg);
    }
}
```

3 Abstract Creator (Factory Method Here)

```
java
abstract class NotificationSender {
    public void notify(String msg) {
        Channel channel = createChannel(); // Factory Method
        channel.send(msg);
    }

    protected abstract Channel createChannel();
}
```

4 Concrete Creators

```
java
class EmailSender extends NotificationSender {
    protected Channel createChannel() {
        return new EmailChannel();
    }
}

class SmsSender extends NotificationSender {
    protected Channel createChannel() {
        return new SmsChannel();
    }
}

class PushSender extends NotificationSender {
    protected Channel createChannel() {
        return new PushChannel();
    }
}
```

24

NOTES

1 The Java solution:

CODE	WHAT IT DOES
<pre>interface Channel { void send(String msg); }</pre>	The product.
<pre>class EmailChannel implements Channel { public void send(String msg) { System.out.println("Email: " + msg); } }</pre>	Concrete products (SmsChannel, PushChannel similar).
<pre>abstract class NotificationSender { public void notify(String msg) { Channel channel = createChannel(); channel.send(msg); }</pre>	The workflow, written once in the creator...
<pre> protected abstract Channel createChannel(); }</pre>	...and the factory method it relies on.
<pre>class EmailSender extends NotificationSender { protected Channel createChannel() { return new EmailChannel(); } }</pre>	Concrete creators decide the channel (SmsSender, PushSender similar).

- 2 `notify()` is the “some operation” of the generic structure: it never names a concrete channel, which is exactly what makes adding WhatsApp a matter of two new classes.

Original slides from the SE322 lecture deck, with study notes. Check exact wording against the [original deck](#). Notes and questions are study aids. Made by Shoug Alomran.