

Software Design and Architecture

[Creational Design Patterns] – Chapter 06, L02

Lecture Outlines

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

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

Factory Method Example

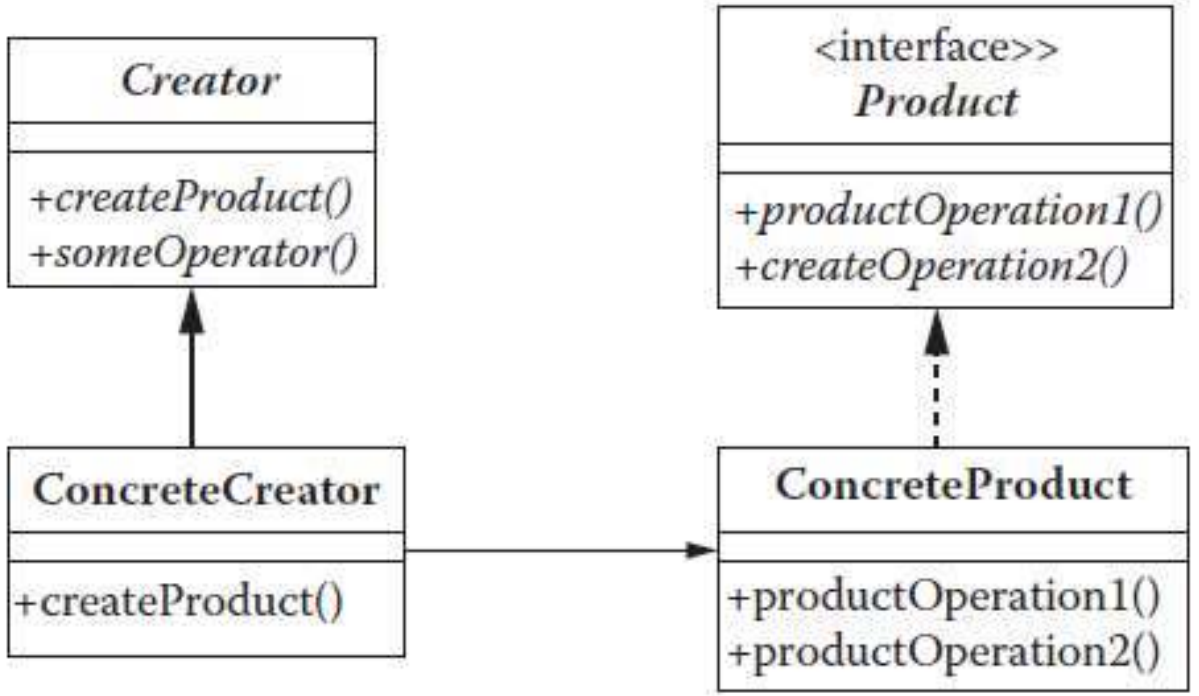
Structure

- 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.



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

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

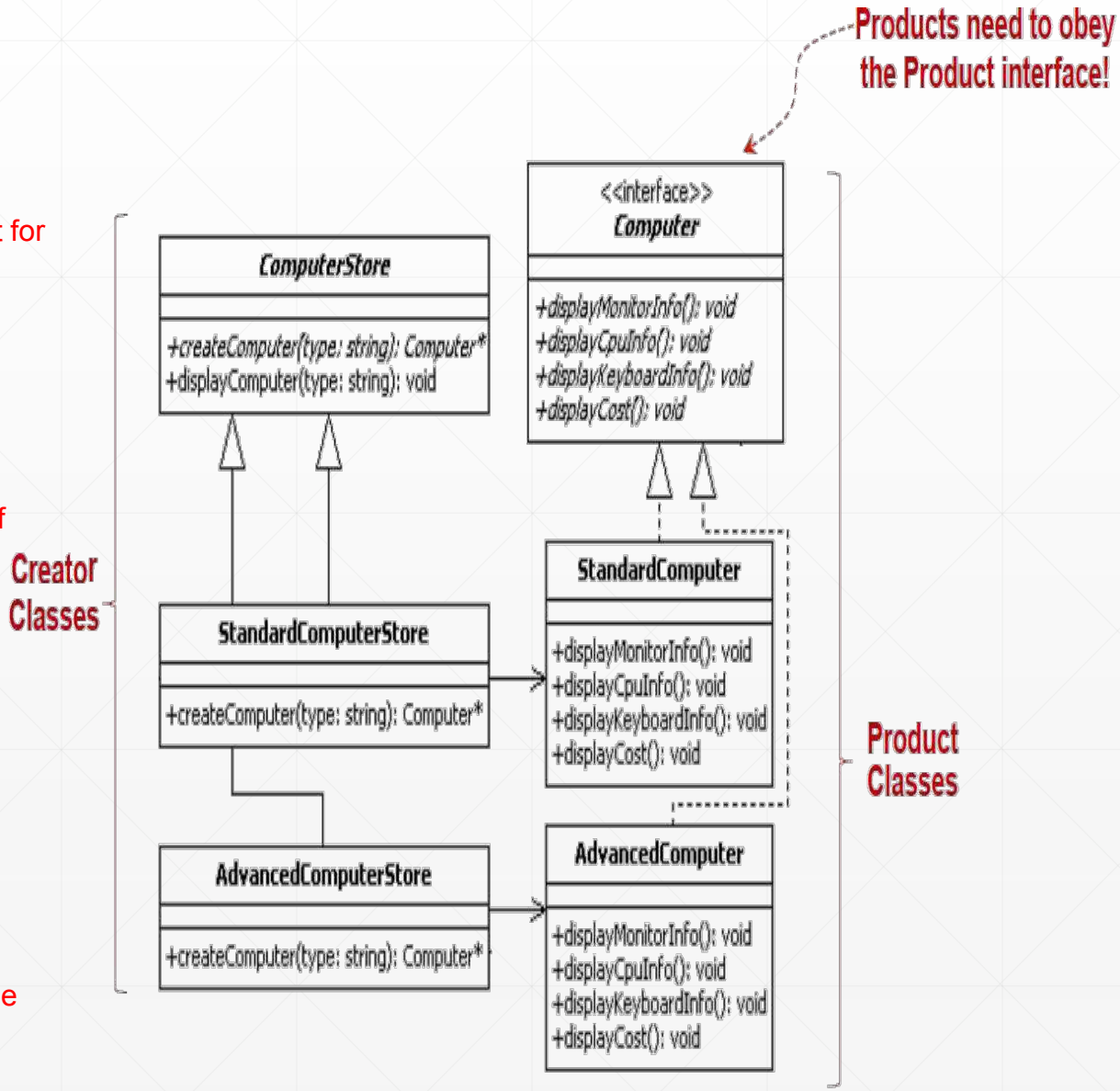
- The `Computer` interface defines the common methods (`displayMonitorInfo()`, `displayCpuInfo()`, etc.) that all computers (standard or advanced) must implement.

4. StandardComputer and AdvancedComputer

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

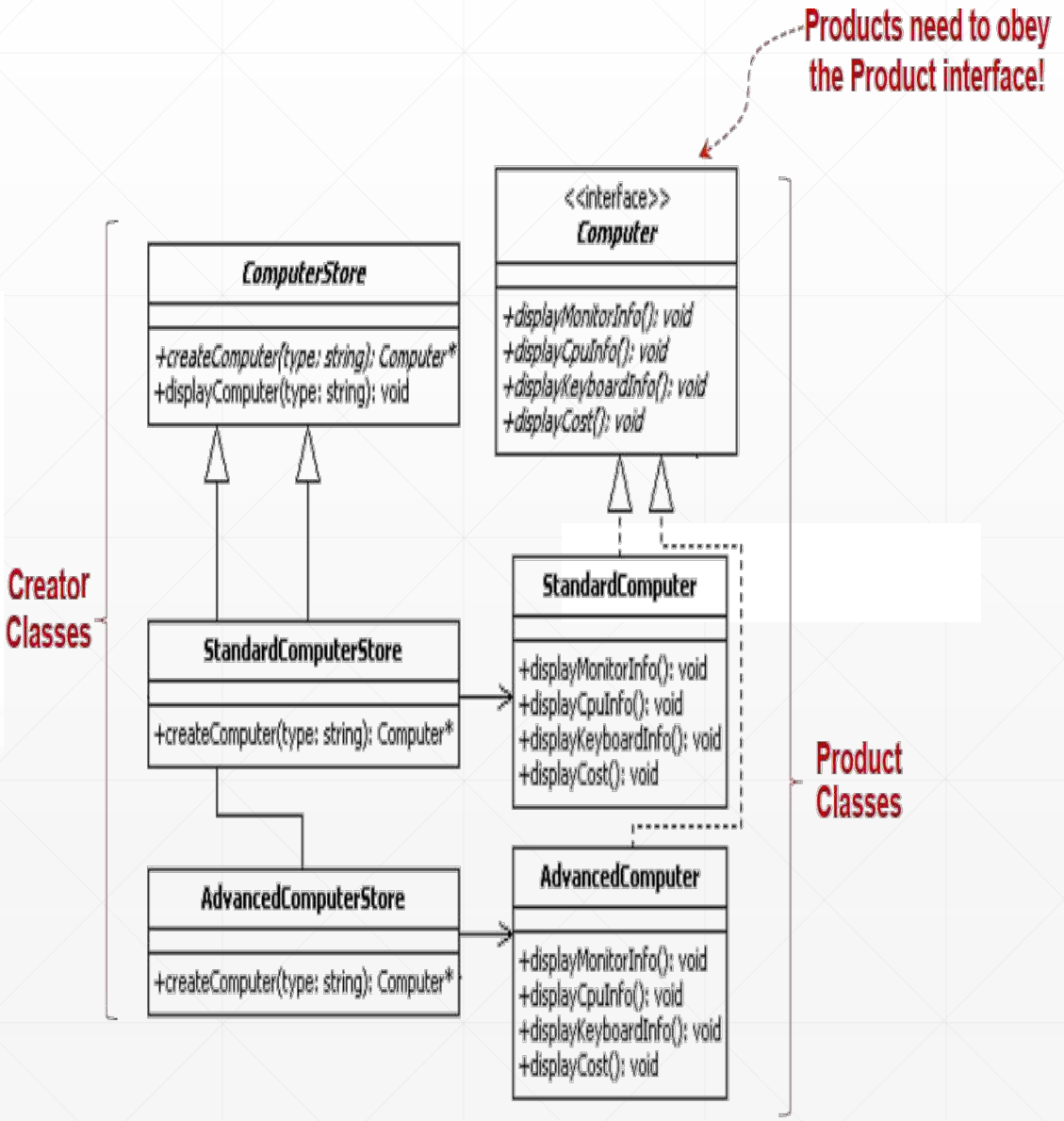
Summary:

- **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.



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.



Factory Method Example

- Sample code for the **ComputerStore** Factory: **in C++**

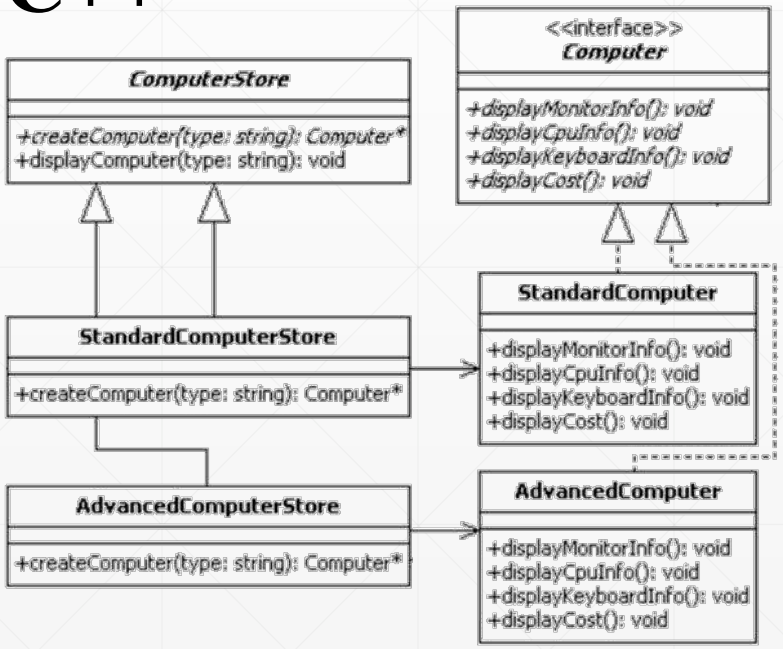
```
#include <string>

// 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).**

Factory Method Example

- Sample code for the **ComputerStore** Factory:

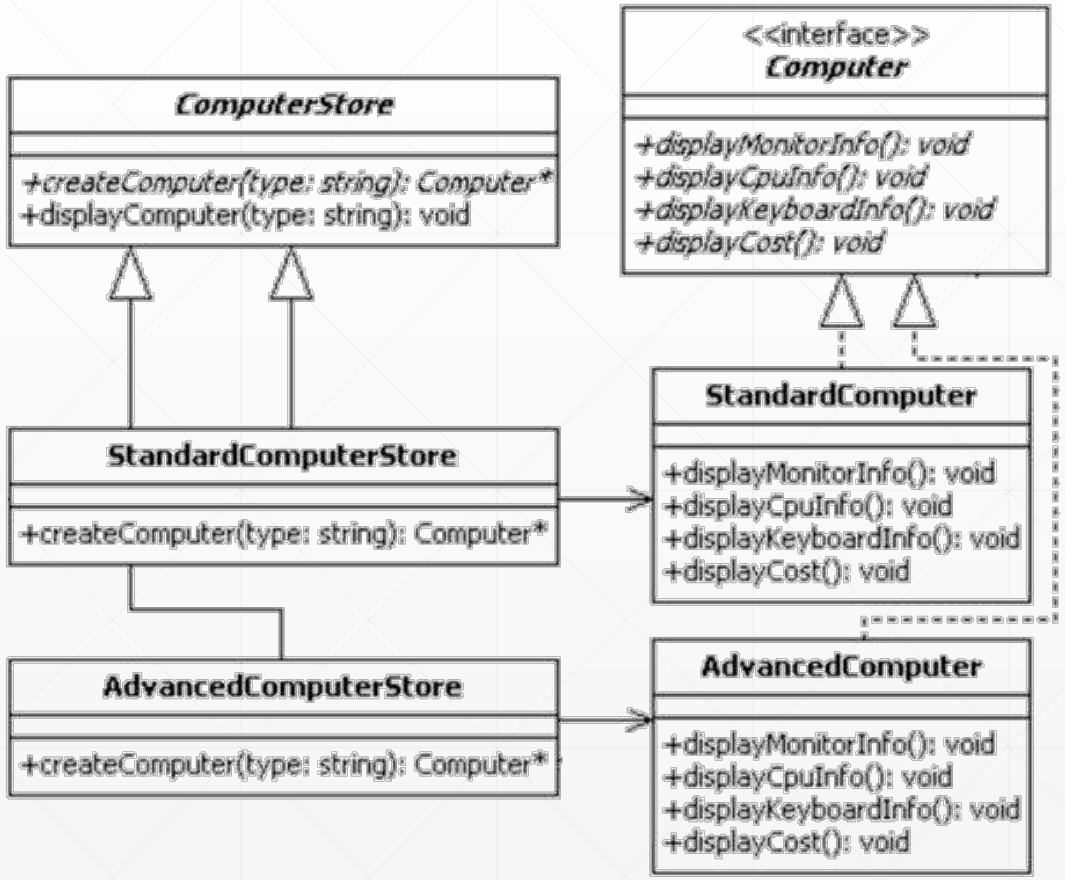
```
#include "Computer.h"

// 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.
}
```



Factory Method Example

- Sample code for the **ComputerStore** Factory:

```
// Implement the factory method.
Computer* StandardComputerStore::createComputer(string type) {

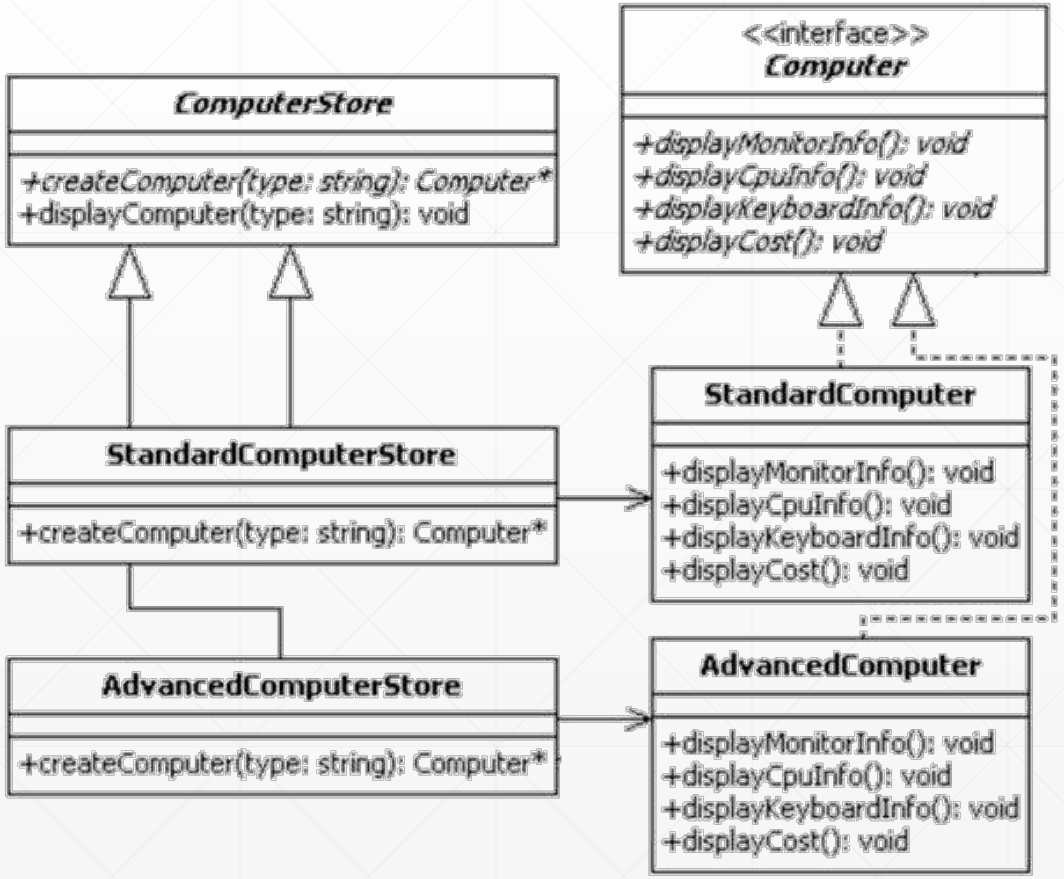
  // Pointer to a computer object.
  Computer* computer;

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

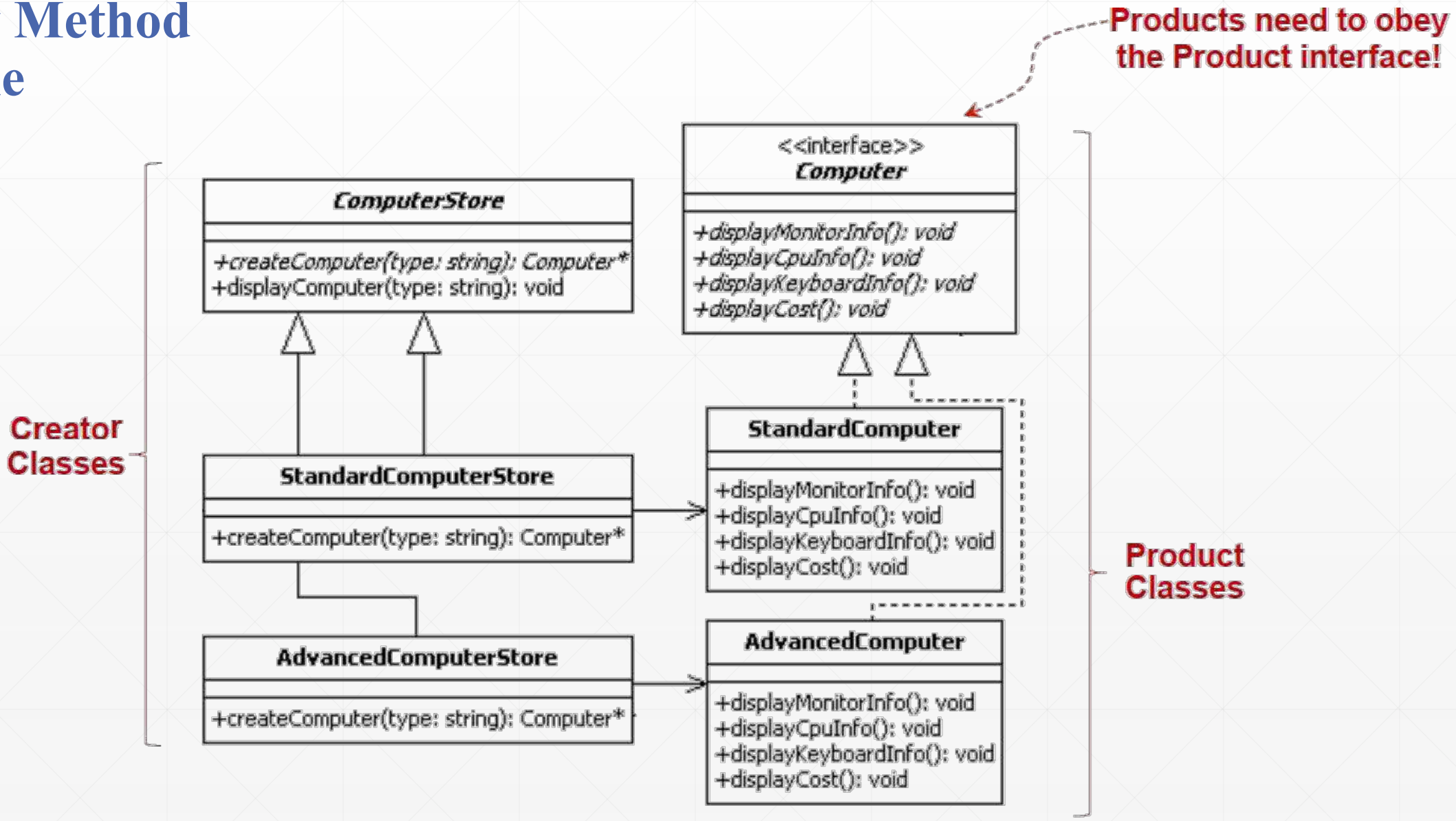
    // Create the StandardComputer. Clients are responsible for
    // cleaning up the memory for the computer object. Internally,
    // StandardComputer uses StandardComputerPartsFactory to create
    // a standard computer.
    computer = new StandardComputer;
  }
  else {

    // 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;
}
```



Factory Method Example



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.

```java
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);   } }
```

Extra

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

```java
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"; }
}
```

```java
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"; }
}
```

```java
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");
    }
}
```

```java
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");
    }
}
```

```java
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");
}
```

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.

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

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**.

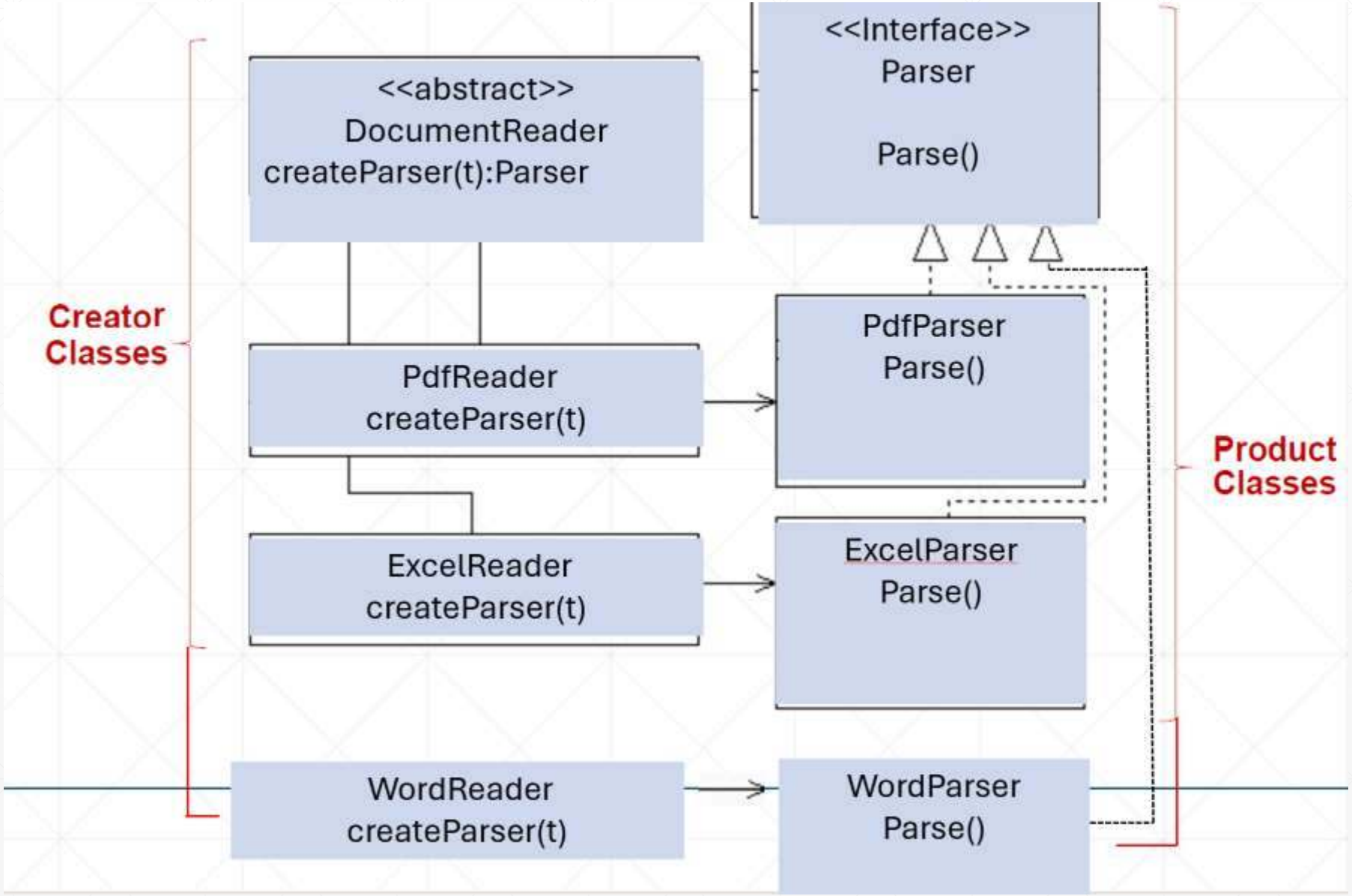
- To support a new type (e.g., PowerPoint, Text), you **add a new reader subclass** that returns the new parser—**no changes to existing client code** are needed.

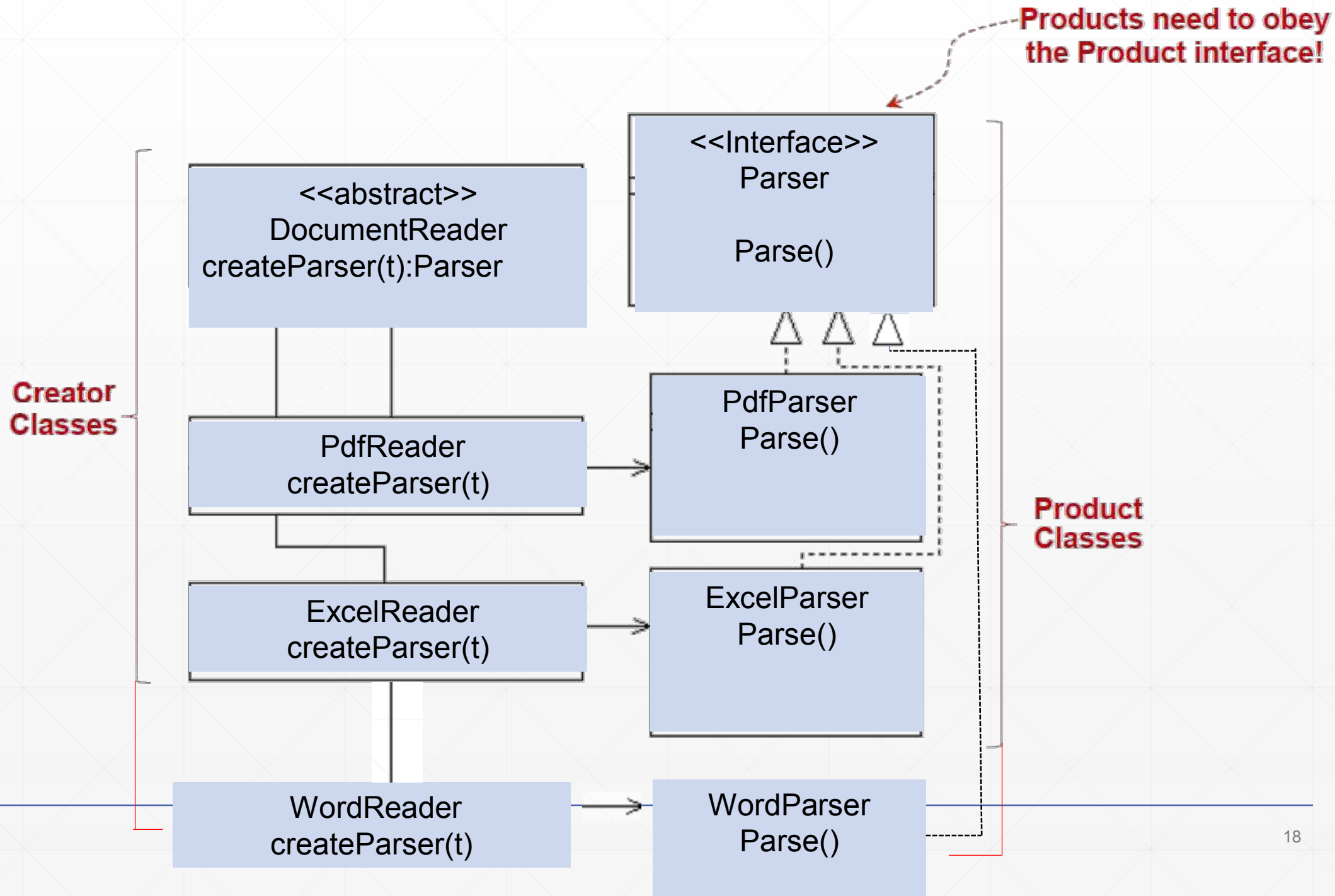
- **Which design pattern is most appropriate here?**

- **Draw a UML diagram for your design?**

- **Write a simple Java code example showing how the Factory Method ?**

Extra:


<<abstract>>
DocumentReader
createParser(t):Parser
<<Interface>>
Parser
Parse()
Creator Classes
PdfReader
createParser(t)
PdfParser
Parse()
ExcelReader
createParser(t)
ExcelParser
Parse()
Product Classes
WordReader
createParser(t)
WordParser
Parse()


Products need to obey the Product interface!
<<Interface>>
Parser
Parse()
<<abstract>>
DocumentReader
createParser(t):Parser
Creator Classes
PdfReader
createParser(t)
PdfParser
Parse()
ExcelReader
createParser(t)
ExcelParser
Parse()
Product Classes
WordReader
createParser(t)
WordParser
Parse()

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.

```java
// 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();
    }
}
```

```java
// 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...");
    }
}
```

```java
// ----- 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();
    }
}
```

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., `Parser` → `PdfParser`, `WordParser`).	Deals with **multiple product hierarchies** that are related (e.g., `Button`, `Checkbox` 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., `createProduct(type)` or `createParser()`).	Client calls multiple factory methods (e.g., `createButton()`, `createCheckbox()`), but all products are guaranteed to belong to the same family.
Example	A `DocumentReader` creates **one parser** (PDF, Word, or Excel).	A `GUIFactory` creates a **set of widgets** (Button + Checkbox) for one platform (Windows, Mac, Linux).

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).

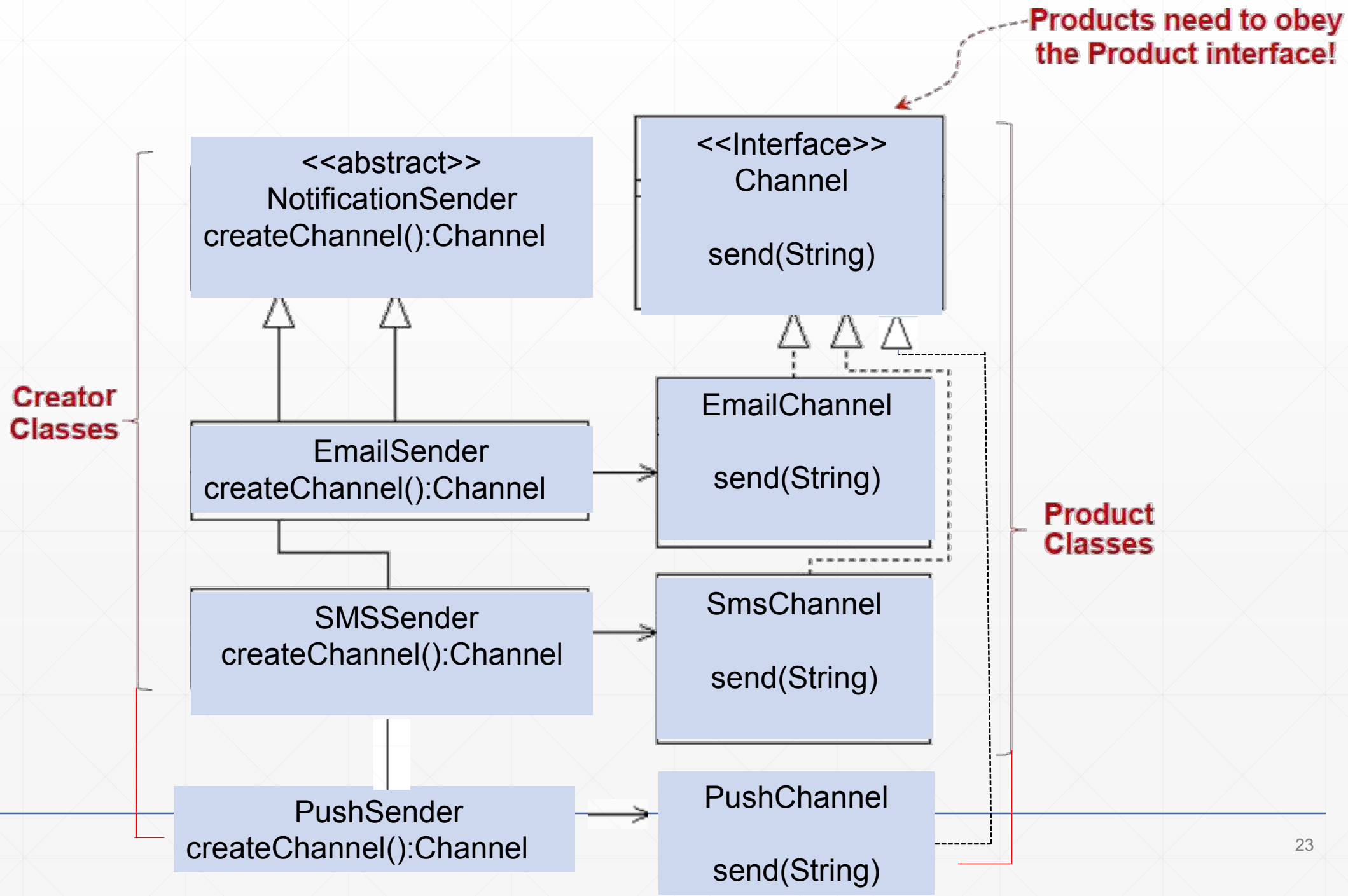
The base class declares **one overridable method** whose job is to return the **Channel** object to use.

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.


Products need to obey the Product interface!
<<abstract>>
NotificationSender
createChannel():Channel
<<Interface>>
Channel
send(String)
Creator Classes
EmailSender
createChannel():Channel
EmailChannel
send(String)
SMSSender
createChannel():Channel
SmsChannel
send(String)
PushSender
createChannel():Channel
PushChannel
send(String)
Product Classes

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();
    }
}
```