2025-2 / Lorenzon : OO - Design Patterns - Factory

Design Patterns – Factory

O Factory Method também faz parte dos padrões criacionais do catálogo GoF. Ele é usado quando queremos delegar a responsabilidade de criação de objetos para subclasses, de modo que o código cliente não precise conhecer diretamente os construtores.

Estrutura da ideia

- Uma interface ou classe abstrata define o contrato para o objeto que será criado.
- Uma classe concreta implementa essa interface.
- Uma fábrica (factory) contém o método que devolve a instância correta. (Pode ser abstrata, delegando às subclasses, ou concreta com simples decisão condicional.)

Vamos iniciar com um código trivial, sem uso de interfaces, numa programação sem uso de Factory:

```
public class Estudante {
    public String getDescricao() {
        return "Sou um estudante.";
    }
}
public class Professor {
    public String getDescricao() {
        return "Sou um professor.";
    }
}
public class App {
    public static void main(String[] args) {
        // O cliente precisa instanciar diretamente:
        Estudante e = new Estudante();
        Professor p = new Professor();

        System.out.println(e.getDescricao());
        System.out.println(p.getDescricao());
    }
}
```

Neste código temos um problema: O cliente (App) conhece e instancia cada classe concreta com new. Se amanhã surgirem outros tipos (Diretor, Funcionario, Secretario...), o App precisará ser modificado em vários pontos. Isso gera alto acoplamento e dificulta manutenção.

Aplicando a ideia do padrão Factory para este código teríamos (Mantendo as classes Estudante e Professor como escritas):

```
// Fábrica centralizando a criação
public class PessoaFactory {
    public static Object criarPessoa(String tipo) {
        if (tipo.equalsIgnoreCase("estudante")) {
            return new Estudante();
        } else if (tipo.equalsIgnoreCase("professor")) {
            return new Professor();
        }
        throw new IllegalArgumentException("Tipo de pessoa inválido: " + tipo);
    }
}

// App de teste
public class App {
    public static void main(String[] args) {
        // Cliente só pede para a fábrica
        Estudante e = (Estudante) PessoaFactory.criarPessoa("estudante");
        Professor p = (Professor) PessoaFactory.criarPessoa("professor");

        System.out.println(e.getDescricao());
        System.out.println(p.getDescricao());
    }
}
```

Por que o casting é necessário? A fábrica não tem um tipo comum para retornar (Pessoa, interface ou superclasse). Então ela devolve Object. Object é a classe mais genérica do Java, mas não sabe nada de métodos como getDescricao(). Para poder chamar getDescricao(), você precisa converter (cast) de volta para o tipo certo.

Como evitar o casting? Criando uma interface ou classe abstrata (Pessoa), aí a fábrica retorna Pessoa e cada classe concreta (Estudante, Professor) implementa getDescricao(). Assim o cliente não faz cast.

A seguir uma versão mais evoluída e recomendada de Factory em Java:

```java
//O contrato
public interface Pessoa {
    String getDescricao();
}
```

```java
//Assinando o contrato
public class Estudante implements Pessoa {
    @Override
    public String getDescricao() {
        return "Sou um estudante.";
    }
}
```

```java
public class Professor implements Pessoa {
    @Override
    public String getDescricao() {
        return "Sou um professor.";
    }
}
```

```java
public class PessoaFactory {
    public static Pessoa criarPessoa(String tipo) {
        if (tipo.equalsIgnoreCase("estudante")) {
            return new Estudante();
        } else if (tipo.equalsIgnoreCase("professor")) {
            return new Professor();
        } else {
            return null;
        }
    }
}
```

```
public class App {
    public static void main(String[] args) {
        // Cliente só pede para a fábrica
        // Não precisa o casting, é resolvido por Polimorfismo e contrato
        Pessoa e = PessoaFactory.criarPessoa("estudante");
        Pessoa p = PessoaFactory.criarPessoa("professor");

        System.out.println(e.getDescricao());
        System.out.println(p.getDescricao());
    }
}
```

Desta forma o cliente não sabe mais como os objetos são criados, só pede à fábrica. Facilita manutenção e extensão (se amanhã quisermos um Funcionario, só mexemos na fábrica, não em todo o código). Reduz acoplamento e centraliza a lógica de criação.

Explicação passo a passo

1. **Contrato (Product)**
   - Pessoa é a *interface* que define o comportamento comum: getDescricao().
   - Objetivo: permitir que o cliente use um tipo abstrato (Pessoa) sem conhecer implementações concretas.
2. **Produtos concretos (ConcreteProduct)**
   - Estudante e Professor implementam Pessoa e fornecem suas versões de getDescricao().
3. **Fábrica (Creator / Factory)**
   - `PessoaFactory` centraliza a lógica de criação.
   - O método criarPessoa(String tipo) decide **qual** implementação concreta instanciar e retorna uma referência do tipo Pessoa.
4. **Cliente (Client)**
   - App solicita objetos à fábrica: PessoaFactory.criarPessoa("estudante").
   - App só conhece Pessoa — chama getDescricao() sem conhecer a classe concreta.
5. **Fluxo de execução**
   - App chama PessoaFactory.criarPessoa("estudante").
   - PessoaFactory verifica o parâmetro e cria new Estudante().
   - Estudante (como Pessoa) é retornado para App.
   - App chama p1.getDescricao() : obtém "Sou um estudante."

6. **Por que isso é melhor que instanciar diretamente?**
    - **Desacoplamento:** App não depende de new Estudante()/new Professor().
    - **Centralização:** mudanças na criação ficam só na fábrica.
    - **Extensão:** adicionar um novo tipo exige apenas: criar nova classe que implemente Pessoa + atualizar a fábrica.

Boas práticas e variações

- **Evitar strings "mágicas"**: use um enum TipoPessoa { ESTUDANTE, PROFESSOR } em vez de String para reduzir erro de digitação;
- **Evitar longos if/else**: use um Map<String, Supplier<Pessoa>> para registrar criadores e evitar muitos ifs;
- **Lançar exceção** (em vez de retornar null) faz erros aparecerem cedo;
- **Factory Method (variante com herança)**: se preferir, torne a fábrica abstrata e crie sub-fábricas especializadas (cada subclasse sabe criar seu produto);
- **Injeção vs. Factory**: para projetos grandes, combine com DI (Spring) para desacoplar ainda mais.

Exemplo rápido de extensão (adicionar Diretor)

```
public class Diretor implements Pessoa {
    @Override
    public String getDescricao() {
        return "Sou um diretor.";
    }
}
```

```
// Na PessoaFactory, só adiciona:
else if ("diretor".equalsIgnoreCase(tipo)) {
    return new Diretor();
}
```

Observações finais.

- O cliente (App) não precisa conhecer new Estudante() ou new Professor();
- Toda a lógica de criação fica encapsulada na **Factory**;
- Podemos facilmente estender para outros tipos (Funcionario, Diretor, etc.) sem modificar o código cliente;
- Se quisermos o **Factory Method clássico** mesmo (com herança), basta tornar PessoaFactory abstrata e criar EstudanteFactory, ProfessorFactory, etc., cada uma retornando seu tipo.