

# Factory

Classificação: Criacional

# Padrões criacionais

- Tratam dos **aspectos da criação de objetos** (instânciação).
- **Encapsulam** a lógica de criação de objetos promovendo a organização do código.
- Tem por objetivo reduzir o **acoplamento** entre as classes clientes e classe instanciada.

# Problema

- Como criar uma **instância** de uma classe sem gerar uma dependência (**acoplamento**) entre a classe cliente e a classe instanciada?
- Como lidar com mudanças na hierarquia de classes?

# Padrão de projeto: Factory

- Classificação: **criacional**
- Um dos padrões mais usados, trata dos detalhes de criação de objetos.
- O Factory é um padrão que encapsula a operação de criação de objetos, fornecendo uma interface que retorna a instância desejada para as classes cliente.
- Vantagem de uso: permite que a classe cliente solicite a criação de objetos sem especificar a classe concreta.

# Operador **new**

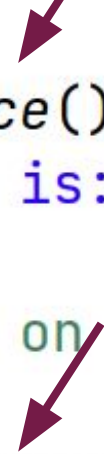
- No java o operador **new** permite criar uma instância (objeto)
- Essa instância é gerada a partir de uma classe **concreta**
- Entretanto, em tempo de codificação, nem sempre **sabemos** qual é a **classe concreta**!
- **Delegar** ao factory a criação de novas instâncias:
  - flexibilidade caso a classe concreta seja modificada
  - controlar, limitar e rastrear a criação de novos objetos

# Exemplo: API Java

O método **getInstance()** do JDK é um exemplo do uso de fábrica para obter uma instância, liberando o cliente da necessidade de gerenciar a criação de objetos e a resolução das dependências necessárias.

```
// Create a calendar object
Calendar cal = Calendar.getInstance();
System.out.println("Date and time is: " + cal.getTime());

// create a currency object based on the user "Country"
Locale lc = Locale.CANADA;
Currency moeda = Currency.getInstance(lc);
System.out.println("Moeda local: " + moeda.getDisplayName());
```




# Código vulnerável a mudanças

## Cenário problemático:

(1) código faz uso de várias classes concretas.

(2) código terá que ser re-escrito a medida que novas classes são adicionadas

Em outras palavras esse código não está finalizado (continua aberto a modificações)



```
switch (formato) {  
  case "jpg":  
    strategy = new JPGSaveImage();  
    break;  
  case "png":  
    strategy = new PNGSaveImage();  
    break;  
  case "gif":  
    strategy = new GIFSaveImage();  
    break;  
  default:  
    strategy = new JPGSaveImage();  
    break;  
}
```

# Identificar as partes que mudam

- **Princípio**: Identificar partes do código que "evoluem" e separar das partes que se mantêm as mesmas.
- **Encapsular** as partes do código que instanciam classes (sujeitas a modificações) separando elas do resto da aplicação.

```
switch (formato) {  
  case "jpg":  
    strategy = new JPGSaveImage();  
    break;  
  case "png":  
    strategy = new PNGSaveImage();  
    break;  
  case "gif":  
    strategy = new GIFSaveImage();  
    break;  
  default:  
    strategy = new JPGSaveImage();  
    break;  
}
```



# Código refatorado usando factory

O código com switch pode ser **refatorado** usando uma **factory**, que passa a ser o responsável em retornar novas instâncias de *"alguma" classe concreta*. Mas de qual classe? Não sabemos e não importa! Essa responsabilidade foi delegada ao factory.

```
switch (formato) {  
  case "jpg":  
    strategy = new JPGSaveImage();  
    break;  
  case "png":  
    strategy = new PNGSaveImage();  
    break;  
  case "gif":  
    strategy = new GIFSaveImage();  
    break;  
  default:  
    strategy = new JPGSaveImage();  
    break;  
}
```



```
SaveImage strategy =  
  factory.createStrategy(formato);
```

# Python: Implementação do **factory**

Exemplo: **Exportar imagem** para outro formato de imagem.

Papéis e responsabilidades:

1. **cliente**: escolhe o arquivo de entrada e o formato de exportação
2. **models**: hierarquia de classes de imagens implementadas
3. **factory**: implementa o método que retorna uma instância de imagem

# Vocabulário

- **Acoplamento:** busca-se baixo acoplamento visando reduzir as dependências entre as classes. Em outras palavras, diminuir o impacto causado ao alterar o código de uma classe nas classes clientes.
- **Encapsulamento:** ocultação da complexidade interna da classe.
- **Hierarquia** de tipos
- **Instância:** objeto criado a partir de um tipo (classe concreta)
- **Responsabilidade:** delegar cada tarefa à classe responsável
- **Refatoração:** melhoria da qualidade do código

# Atividade 1: Factory formas de pagamento

- Implementar a **FactoryPagamento** responsável por criar instâncias de classes concretas de formas de pagamento.
- Somente a factory sabe os tipos de Pagamentos.
- Código deve estar organizado na models, factory e client.

## Atividade 2: implementar Factory NPC

- Factory controla a **criação** de NPCs num jogo. Novos NPCs podem ser adicionados na hierarquia de classes sem quebrar o código nas classes clientes. Proponha 3 tipos de NPCs.