




Programação Orientada a Objetos

Aula 6 – Interfaces; Tipos Genéricos;
Padrões de Projeto



Prof Ronald Chiesse de Souza
ronaldsouza@dcc.ufrj.br



Interfaces

Interfaces

- Outro paradigma para definir ***tipos de dados*** e ***inter-relações entre objetos***, muito semelhante a classes abstratas.
- Também ***não*** podem ser instanciadas, mas seus métodos são ***necessariamente abstratos***. Ou seja, interfaces não suportam implementações padrão.
- Ao contrário de relações de herança, uma classe pode implementar ***várias interfaces***!

Interfaces x Herança/Classe Abstrata

| Interface | Classe Abstrata/Herança |
|---------------------------------------|--|
| Uma classe pode implementar várias | Uma classe só pode herdar de uma |
| Só pode ter métodos abstratos | Pode ter métodos abstratos e concretos |
| Não pode conter métodos estáticos | Pode conter métodos estáticos |
| Atributos somente public static final | Pode ter qualquer tipo de atributo |
| É vista como um contrato | Usada somente com relação “é um” |

Declaração de interface em Java

```
public interface Printavel {  
  
    public abstract String  
    retornaString();  
  
}
```

```
public class Gerente extends Funcionario  
implements Printavel {  
  
    // atributos específicos e outros métodos  
  
    @Override  
    public String retornaString() {  
        return "Gerente " + this.nome + " com  
CPF " + this.cpf;  
    }  
}
```

Tipos Genéricos

Tipos Genéricos

- Tipos genéricos são usados em classes, interfaces, atributos e métodos para deixar uma funcionalidade adaptável a diferentes tipos de dados sem a necessidade de reescrever a funcionalidade
- Pode ser vista como um “template” que define o tipo de dado usado somente em tempo de execução
- Principais vantagens de tipos genéricos: **Cconsolidação de código**, pois o código usando tipos genéricos é definido apenas uma vez; e **reutilização de código**, pois o código não depende de um tipo específico, podendo ser reutilizado com outros não previstos inicialmente.

Exemplo de instanciação com tipo genérico

- Classe ArrayList, para criar um array de tamanho variável

```
public class Main {  
  
    public static void main(String[] args) {  
        ArrayList<String> lista_strings = new ArrayList<String>();  
  
        lista_strings.add("Frase 1");  
        lista_strings.add("Frase 2");  
        lista_strings.add("Frase 3");  
  
        System.out.println(lista_strings);  
    }  
}
```


Criando interfaces com tipos genéricos

```
import java.util.ArrayList;

public interface Ordenavel<T>
{
    public abstract ArrayList<T>
ordena();
}
```

```
public class ControleFuncionario implements
Ordenavel<Funcionario>{
    ArrayList<Funcionario> lista;

    @Override
    public ArrayList<Funcionario> ordena() {
        Collections.sort(this.lista,
Comparator.comparing(func -> func.getNome()));
        return this.lista;
    }
}
```

Padrões de Projeto

Definição

- Designs genéricos aplicáveis em muitos problemas, mas que exigem adaptação a cada caso particular de cada programa.
- Não são soluções prontas, mas sim **técnicas** que podem ser usadas para organizar e facilitar a programação.
- Aumentando a clareza da comunicação entre desenvolvedores.
- Veremos três padrões: Singleton, Factory e Observer

Padrões de projeto que veremos

- Singleton \Rightarrow Classe com uma única instância usada por toda a aplicação
- Factory \Rightarrow Classe usada como interface para instanciar diferentes classes
- Observer \Rightarrow Classe que “escuta” atualizações em determinada parte do sistema e notifica outra parte ao detectar alguma modificação

Singleton - Conceito

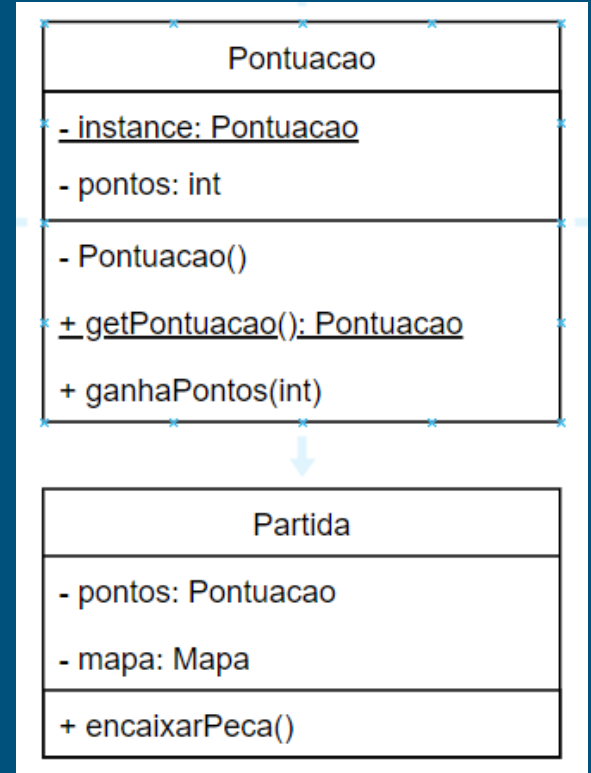
- Um Singleton é uma classe que só pode ter uma instância, que é compartilhada com toda a aplicação.
- Normalmente ele contém atributos e métodos que podem ser utilizados em todo o programa, como uma variável global.
- Seu construtor é feito de forma que instanciar mais de um objeto é impossível.

Exemplos de Singleton

- Conversor de medidas em um sistema meteorológico
- Configurações de aplicativo
 - Usamos o singleton para armazenar todo tipo de configuração que o usuário possa customizar em uma só estrutura acessível para todo o programa (tema escuro, idioma, etc)
- Pontuação de jogo
- Classe para salvar log
- Classe para conectar com APIs externas / banco de dados

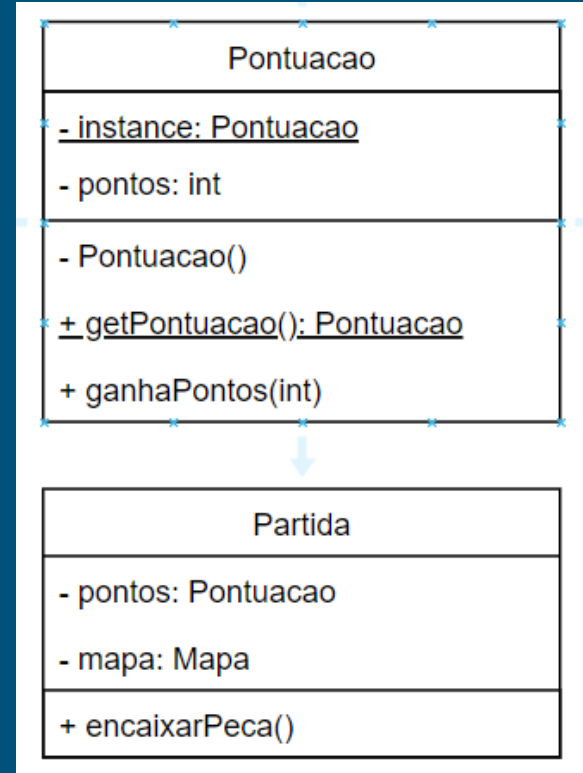
Pontuação em jogo (Singleton)

- Estamos desenvolvendo um jogo de Tetris.
- Digamos que nossa pontuação é cumulativa, então usamos um Singleton para armazenar a pontuação.
- A cada linha fechada em uma partida, chamamos o método ganhaPontos do Singleton de Pontuação.



Pontuação em jogo (Singleton) - Código

```
public class Pontuacao{  
    private static Pontuacao instance;  
  
    private Pontuacao(){  
        // método construtor privado  
    }  
  
    public static Pontuacao getPontuacao(){  
        if (instance == null)  
            instance = new Pontuacao();  
        return instance;  
    }  
  
    // outros métodos  
}
```



Pontuação em jogo (Singleton) - Código

```
public class Pontuacao{
    private static Pontuacao instance;

    private Pontuacao(){
        // método construtor
    }

    public static Pontuacao getInstance(){
        if (instance == null){
            instance = new Pontuacao();
        }
        return instance;
    }

    // outros métodos
}
```

```
public class Partida{
    private Pontuacao pontos;

    public Partida(){
        this.pontos = Pontuacao.getInstance();
    }

    public void encaixaPeca(){
        // código antes
        if (fechou_linha){
            this.pontos.ganhaPontos(qtde_linhas);
        }
        // código depois
    }
}
```

Singleton - Observações

- O construtor do Singleton precisa ser privado. Ele deve ser instanciado apenas pelo chamado pelo método de obter a instância (getConversor).
- Se nosso aplicativo precisar de novos atributos, podemos usar a mesma classe, apenas implementando uma nova função para cada novo atributo.
- Sua centralização evita que erros de implementação sejam espalhados pelo código.

Factory



Factory - Conceito

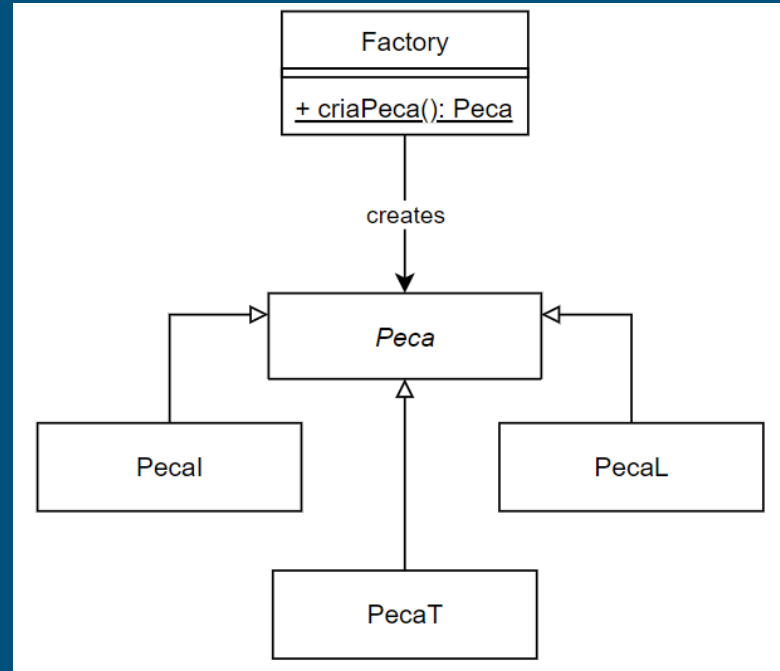
- Como o próprio nome já diz, esse padrão tenta simular uma fábrica, ou seja, ser usado na criação de objetos.
- Seu principal objetivo é criar uma interface que outras partes do programa consigam usar para instanciar classes, ***sem usar diretamente os construtores.***
- Isso permite que todas as criações de objetos possam ser feitas na mesma chamada de função, e que objetos distintos possam ser retornados numa mesma função.

Exemplos de Factory

- Criação de diferentes tipos de itens em um jogo com inventário
- Criação das peças em um jogo (ex. Tetris)
- Conexão com diferentes tipos de bancos de dados
- Criação de componentes de interface gráfica (*Graphic User Interface* - GUI)

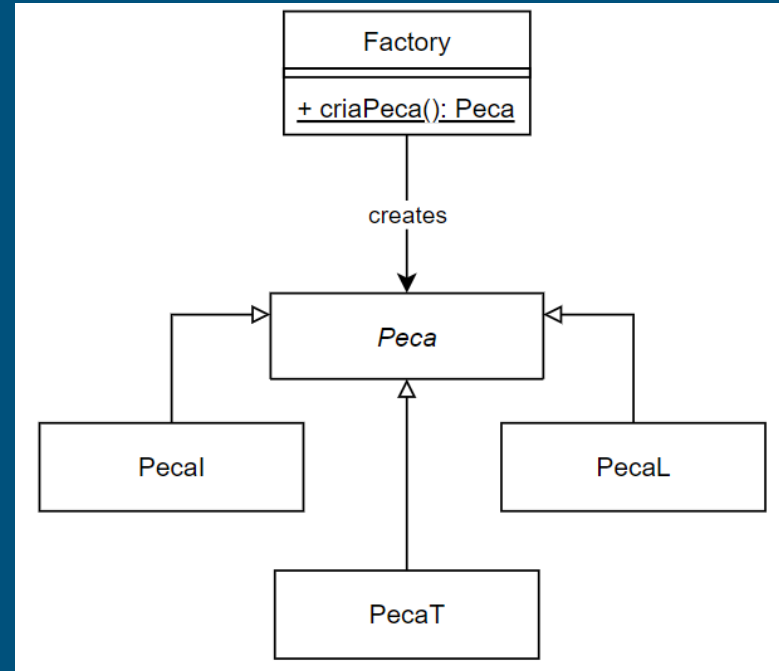
Criação de peças em jogo (Factory)

- Continuando o jogo de Tetris
- Colocamos uma fábrica para criar os diferentes tipos de peças



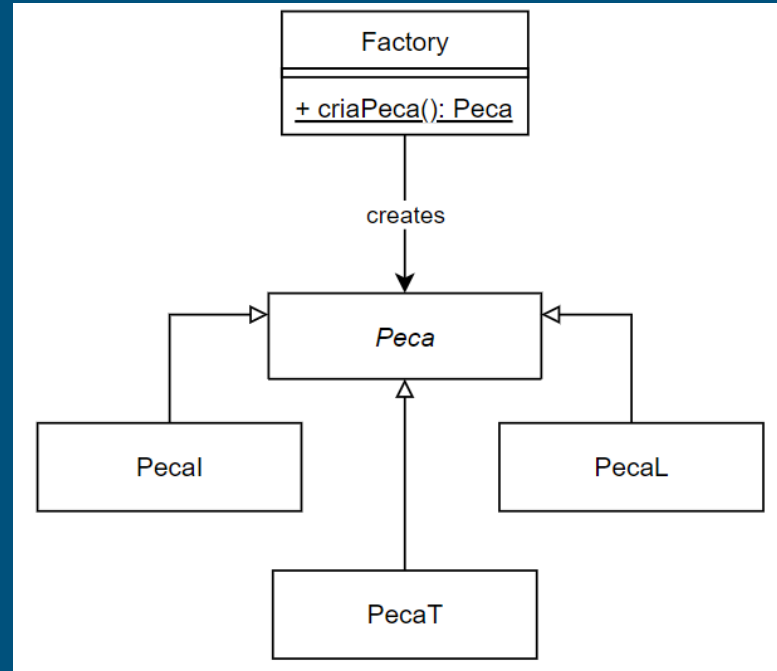
Criação de peças em jogo (Factory) - Código

```
public class Factory{  
    public static Peca createPeca(String tipo){  
        Peca peca_retorno;  
        if(tipo.equals("PecaI")) {  
            PecaI peca = new PecaI();  
            peca_retorno = peca;  
        }  
        if(tipo.equals("PecaT")) {  
            PecaT peca = new PecaT();  
            peca_retorno = peca;  
        }  
        if(tipo.equals("PecaL")) {  
            PecaL peca = new PecaL();  
            peca_retorno = peca;  
        }  
        return peca_retorno;  
    }  
}
```



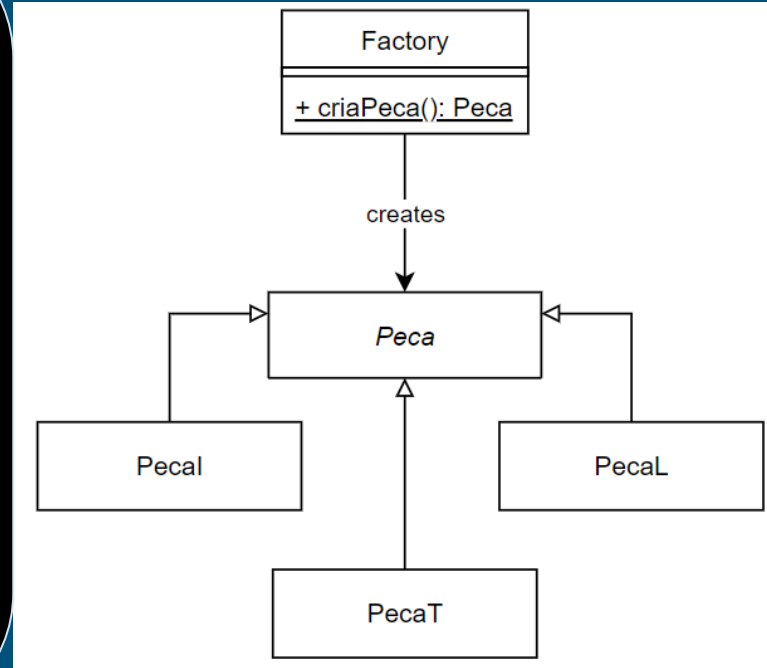
Criação de peças em jogo (Factory)

- Continuando o jogo de Tetris
- Colocamos uma fábrica para criar os diferentes tipos de peças de forma aleatória



Criação de peças em jogo (Factory) - Código

```
public class Factory{  
    public static Peca createPeca(){  
        Peca peca_retorno = null;  
        Random rand = new Random();  
        int num = rand.nextInt(3);  
        switch (num){  
            case 0: peca_retorno = new PecaI();  
                    break;  
            case 1: peca_retorno = new PecaT();  
                    break;  
            case 2: peca_retorno = new PecaL();  
                    break;  
        }  
        return peca_retorno;  
    }  
}
```



Factory - Observações

- Com uma classe factory, os construtores originais das classes não devem ser usados.
- Caso desejemos uma nova peça, podemos continuar usando a mesma função para criação de todas, minimizando a mudança de código necessária.
- Se quisermos ter uma geração aleatória de itens, não precisamos de uma estrutura condicional complexa, pois temos uma mesma função para todas as peças.