Computação II – Orientação a Objetos

Fabio Mascarenhas - 2016.2

http://www.dcc.ufrj.br/~fabiom/java

Classes abstratas

- Até agora, usamos *interfaces* toda vez que queríamos representar algum conceito abstrato em nosso programa, não importa a forma como ele era implementado

- Programar com interfaces é flexível, mas a restrição de só podermos ter métodos em uma interface às vezes é inconveniente, e pode levar a duplicação de código

- Implementações *default* de métodos em interfaces surgiram apenas em Java 8, mas não elimina toda a duplicação

- Para contornar isso, Java oferece um segundo mecanismo para representar objetos abstratos: as *classes abstratas*

Classes abstratas - sintaxe

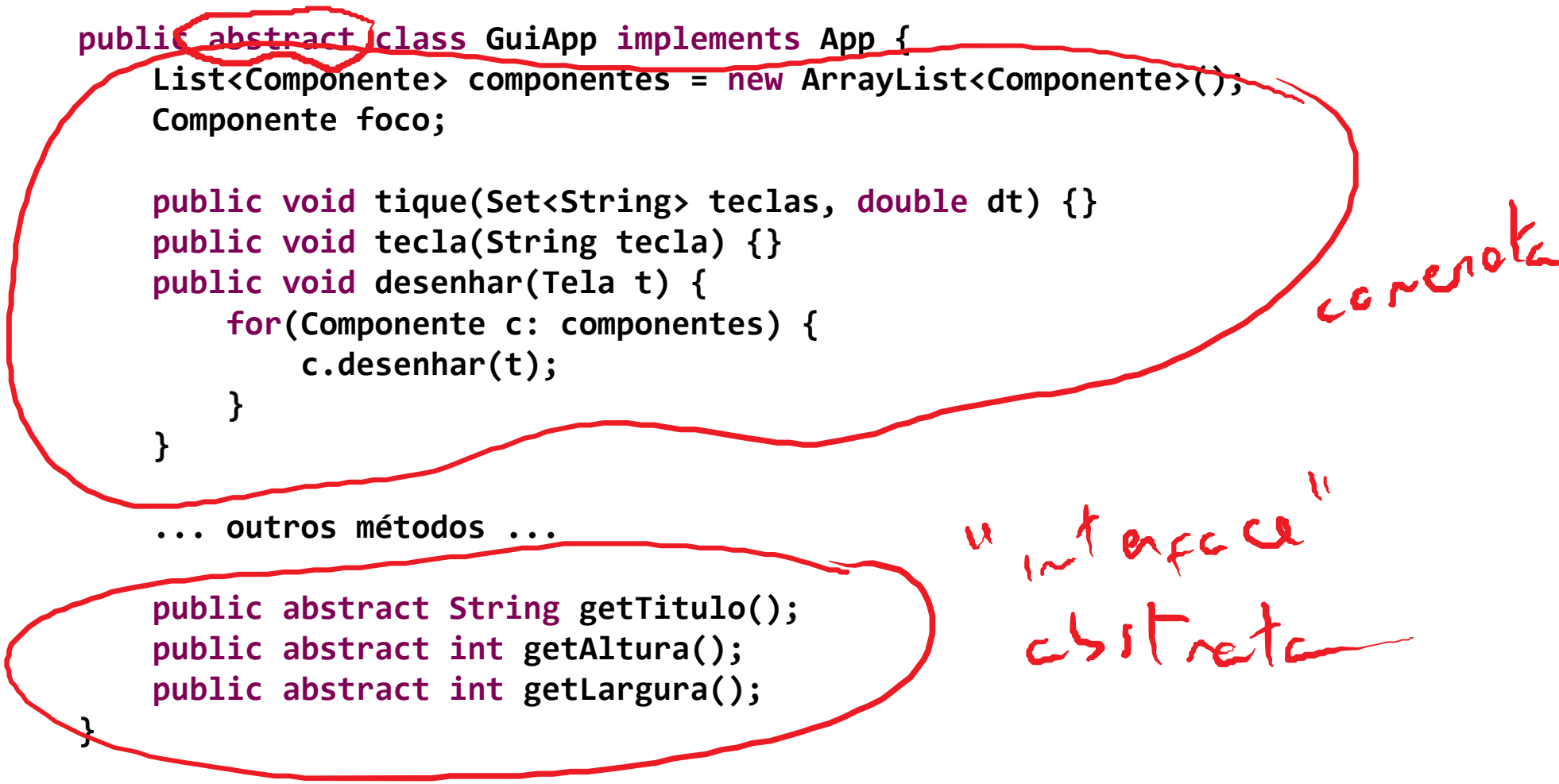
- Uma classe abstrata é declarada com a palavra-chave `abstract` acompanhando `class`:

```
public abstract class GuiApp implements App {
    List<Componente> componentes = new ArrayList<Componente>();
    Componente foco;

    public void tique(Set<String> teclas, double dt) {}
    public void tecla(String tecla) {}
    public void desenhar(Tela t) {
        for(Componente c: componentes) {
            c.desenhar(t);
        }
    }

    ... outros métodos ...

    public abstract String getTitulo();
    public abstract int getAltura();
    public abstract int getLargura();
}
```



Classes abstratas – métodos abstratos

- Uma classe abstrata pode ter campos e métodos, como qualquer outra classe, mas não podemos instanciar uma classe abstrata, mesmo se ela tem construtores públicos

- Em troca, podemos declarar métodos sem corpo, do mesmo modo que em uma interface, basta declará-los como `abstract` também

- Podemos até implementar uma interface, e deixar métodos em aberto, sem declará-los! É como se os declarássemos `abstract`

Usando classes abstratas - herança

- Se não podemos instanciar uma classe abstrata diretamente, para ter instâncias dela criamos uma classe concreta que *herda* da classe abstrata

```
public class Calculadora extends GuiApp {
  public Calculadora() {
    componentes.add(new Botao(0, 100, 100, 100, "7", Cor.BRANCO, Cor.PRETO,
                            new AcaoDigito(this, 7)));
    ... outros componentes ...
    componentes.add(new CaixaTexto(0, 0, 400, 100, Tela.CENTROY | Tela.DIREITA,
                            this::getTextoDisplay, Cor.BRANCO, Cor.BRANCO, Cor.PRETO));
  }

  ... outros métodos ...

  public static void main(String[] args) {
    new Motor(new Calculadora());
  }
}
```

Herança

- A relação de herança (`extends`), como `implements`, também é uma relação “é-um”

- Uma instância de `Calculadora` *é uma* instância de `GuiApp`

- As relações “é um” são transitivas: uma instância de `Calculadora` é uma instância de `GuiApp` (via herança), e uma instância de `GuiApp` é uma instância de `App` (via implements), então uma instância de `Calculadora` é uma instância de `App`

- A diferença da herança é que nela a classe também herda a *forma* da outra, e quaisquer implementações de suas operações, não apenas as assinaturas

Herança - restrições

- Uma classe concreta precisa fornecer implementações para todos os métodos abstratos que ela herdou
  - Como uma classe abstrata pode ela própria ter herdado métodos, essa obrigação é transitiva
- Uma classe pode implementar quantas interfaces ela quiser, mas só pode herdar de uma única outra classe
- Construtores não são herdados diretamente

Herança - construtores

- Construtores não são herdados diretamente, mas estão disponíveis
- Os construtores da *subclasse* devem chamar um dos construtores da *superclasse* na primeira linha
- Chamamos um construtor da superclasse com o comando `super(...)`, passando os argumentos para o construtor entre os parênteses
- Se não chamamos nenhum construtor, é como se chamássemos o construtor padrão com `super()`, então a superclasse precisa ter um desses!

class Foo { ... sem construtor ... }

public Foo() { super(); }

Template Method

- Um padrão de programação comum em classes abstratas, onde métodos da classe abstrata delegam parte do seu comportamento para as subclasses através de métodos abstratos

- Os métodos de uma classe abstrata podem chamar métodos abstratos dessa classe sem problemas; como uma subclasse concreta precisa fornecer implementações para os métodos abstratos, eles “estarão lá” quando necessário

- O uso desse padrão é comum em *frameworks*: ao invés da aplicação implementar uma interface, ela estende uma classe abstrata e fornece os métodos que faltam