Computação II – Orientação a Objetos

Fabio Mascarenhas - 2014.1

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 assinaturas de métodos em uma interface às vezes é inconveniente, e pode levar a duplicação de código

- 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`:

```
abstract class FiguraPt {
  int x;
  int y;

  public void mover(int dx, int dy) {
    x += dx;
    y += dy;
  }

  public abstract void desenhar(Canvas c);
  public abstract void escala(double f);
  public abstract boolean dentro(int x, int y);
}
```

Classes abstratas – métodos abstratos

- Uma classe abstrata pode ter campos e métodos, como qualquer outra classe, mas não podemos instanciar uma classe abstrata, não importa 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` também.

```
abstract class FiguraPt implements Figura {
  int x, y;

  public void mover(int dx, int dy) {
    x += dx; y += dy;
  }
}
```

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

```
class Retangulo extends FiguraPt {
  int largura;
  int altura;

  public Retangulo(int x, int y, int largura, int altura) {
    this.x = x; this.y = y; this.largura = largura; this.altura = altura;
  }

  public boolean dentro(int x, int y) {
    return (x >= this.x)&&(x <= this.x + largura)&&(y >= this.y)&&(y <= this.y + altura);
  }

  public void escala(double f) {
    altura *= f;
    largura *= f;
  }

  public void desenhar(Canvas c) {
    c.retangulo(x, y, largura, altura, 1, 1, 1);
  }
}
```

Herança e Subtipagem

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

- Uma instância de `Retangulo` *é uma* instância de `FiguraPt`

- As relações “é um” são transitivas: uma instância de `Retangulo` é uma instância de `FiguraPt` (via herança), e uma instância de `FiguraPt` é uma instância de `Figura` (via implements), então uma instância de `Retangulo` é uma instância de `Figura`

- 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!

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

Redefinição de métodos

- A subclasse não está restrita a só fornecer construtores e implementações para métodos abstratos
- Ela também pode *redefinir* métodos, dando uma outra implementação para eles
- Uma redefinição tem a mesma assinatura do método que está sendo redefinido
- Instâncias da subclasse usam sempre a nova implementação do método, *mesmo que este esteja sendo chamado por uma referência para a superclasse*

↳ polimorfismo

Herança de classes concretas e `Object`

- A superclasse que passamos para a cláusula `extends` não precisa ser uma classe abstrata

- Pode ser uma classe qualquer, assim podemos criar novas versões de uma determinada classe, mas que reaproveitam parte do seu código

- Quando não damos uma cláusula `extends`, implicitamente estamos herdando de uma classe embutida chamada `Object`

- `Object` tem um construtor sem parâmetros, por isso nunca percebemos que estávamos herdando dela!

toString, equals, hashCode

- `Object` tem alguns métodos que são úteis quando redefinidos
- O método `toString` é chamado toda vez que Java precisa converter um objeto para uma string (na concatenação com uma string, por exemplo)
- O método `equals` compara um objeto com outro, e devemos redefini-lo quando queremos comparar objetos por estrutura e não por referência
- O método `hashCode` é usado pela coleção `HashMap` como um “resumo” da esturuta de um objeto; se redefinimos `equals`, precisamos redefinir `hashCode` para dois objetos “iguais” terem o mesmo `hashCode`, ou quebramos `HashMap` com esses objetos como chaves

Redefinição com extensão

- Quando redefinimos um método em uma subclasse, podemos chamar o método que está sendo redefinido

- É como se estivéssemos estendendo o método, fazendo tudo o que ele faz, mais algum comportamento extra

- Para chamar o método que está sendo redefinido também usamos `super`, mas em uma chamada de método:

```
public String toString() {
  return "Meu objeto (" + super.toString() + ")";
}
```

Herança e visibilidade

- Campos e métodos privados **não** são visíveis para as subclasses, e métodos privados não podem ser redefinidos

- Existe um terceiro nível de visibilidade dado pela palavra-chave `protected`, que torna um campo ou método *público* para subclasses, mas *privado* para outras classes

- O uso protected é comum quando temos membros que queremos que subclasses acessem, mas não façam parte da interface pública do objeto

```
abstract class ExpressaoBinaria implements Expressao {
    Expressao esq;
    Expressao dir;

    ...

    protected abstract double op(double x, double y);
}
```

Herança vs. Composição

- Herança implica uma relação “é um” entre a subclasse e a superclasse, então ela deve ser evitada se essa relação seria espúria

- Nesse caso, podemos ter o mesmo reaproveitamento de código que temos na herança, com um pouco mais de burocracia, usando composição e delegação

- Cuidado com a literatura introdutória de OO, ela está cheia de exemplos espúrios do uso de herança, como “Círculo extende Elipse”

- Os princípios de projeto SOLID nem sempre podem ser seguidos, mas devem ser o ideal