Interfaces

Programação 2 - Aulas 17 e 18

Objetivos da seção

- Apresentar o conceito de Tipo Abstrato de Dado (“interface”, em Java)
- Mostrar como interfaces permitem o desacoplamento entre interface e implementação
- Mostrar como interfaces criam uma hierarquia de tipos, e permite polimorfismo

Saber quais métodos escrever, com que parâmetros e com que valor de retorno é uma *arte* que os alunos vão adquirir com experiência

Mas existem regras que devem ser aprendidas bem cedo!

Iremos aprender regras de ouro neste módulo!

Definição

- Levando o conceito de classes abstratas ao extremo, chegamos perto das interfaces
- Interfaces definem os métodos com seus parâmetros e tipos de retorno, mas sem implementação alguma
- Quando estamos trabalhando com uma dada interface sabemos quais são os métodos que podemos chamar, o que eles recebem como parâmetros de entrada e o que ele retorna
  - Mas não sabemos como foi implementado
  - Não há implementação associada
  - É como se fosse um acordo de uso (um protocolo)
- Vejamos a implementação da interface FiguraGeometrica

```
public interface FiguraGeometricaPlana extends
```

```
Comparable<FiguraGeometricaPlana> {

   /**
    * Retorna a área da figura geométrica.
    */
   public double getArea();

  /**
    * Retorna o perímetro desta figura geométrica.
    * @return  O perímetro desta figura geométrica.
    */
   public double getPerimetro();

   /**
    * Desenha a figura geométrica na tela.
    */
   public void desenha();
```

- Note que temos uma palavra reservada nova: `interface`
- Como não existe implementação, não é possível instanciar esta interface (usando *new*)
- O que fizemos até agora foi apenas **definir um tipo**
  - **Isso só é útil se fizermos mais duas coisas**
    i) Fornecer uma ou mais implementações para esta interface
    ii) Usar o tipo definido (e não suas implementações ao definir as nossas referências a objetos)
- Vamos então criar uma classe que implementa esta interface?

```
package p2.exemplos;


public class Circunferencia implements FiguraGeometricaPlana {

   private Ponto2D centro;
   private double raio;

   public Circunferencia() {
```

```
        centro = new Ponto2D(0, 0);
        raio = 1;
    }

    public Circunferencia(Ponto2D centro, double raio) throws Exception {
        if (raio <= 0.0)
            throw new Exception("O raio de um circulo nao pode ser nulo!");
        this.centro = centro;
        this.raio = raio;
    }

    public void desenha() {
        System.out.println("Desenhando um circulo.");
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Circunferencia) {
            return false;
        }
        Circunferencia c = (Circunferencia) obj;
        return getCentro().equals(c) && getRaio() == c.getRaio();
    }

    public double getArea() {
        return Math.PI * raio * raio;
    }

    @Override
    public String toString() {
        return "Circulo de raio " + getRaio() + " e centro "
                + getCentro().toString() + ".";
    }

    public Ponto2D getCentro() {
        return centro;
    }
```

```
    public void setCentro(Ponto2D centro) {
        this.centro = centro;
    }

    public double getRaio() {
        return raio;
    }

    public void setRaio(double raio) {
        if (raio > 0.0)
            this.raio = raio;
    }

    public double getPerimetro() {
        return 2 * Math.PI * raio;
    }

    public int compareTo(FiguraGeometricaPlana outra) {
        return ((int)this.getArea() - (int)outra.getArea());
    }
}
```

- Veja como é simples implementar uma interface (parece muito com estender uma classe)
  - Só mudou a cláusula **`implements FiguraGeometricaPlana`** que significa que esta classe implementa a interface FiguraGeometricaPlana
  - **`implements`** é uma palavra reservada
  - **Isso nos obriga a implementar cada método que pertence à interface**
    i) O compilador ajudará você a garantir isso
- Qualquer objeto da classe Circunferencia pode ser tratado como se fosse do tipo FiguraGeometricaPlana
- Observe também que há métodos implementados pela classe que não fazem parte da interface (quais, por exemplo?)
- Agora vamos criar um programa que desenha uma circunferência

```
package p2.exemplos;
```

```
public class DesenhaFiguras {

   public static void desenha( FiguraGeometricaPlana figura ) {
      figura.desenha();
   }

   /**
    * @param args
    */
   public static void main(String[] args) {
      FiguraGeometricaPlana figura = new Circunferencia();
      desenha(figura);
   }

}
```

Manutenção de programas

- Vamos fazer “manutenção” no nosso programa
  - Manutenção é uma atividade extremamente comum feita por programadores
  - Software não é uma coisa estática que não muda depois de feita
  - Há *sempre* mudanças a fazer em programas que são utilizados
  - Programas que não precisam mudar mais é porque ninguém os está utilizando
- Nosso problema de manutenção: o usuário deseja manipular novos tipos de figuras geométricas
  - Para simplificar nosso trabalho, vamos manipular retângulos (isso é simples!)
- A primeira coisa que fazemos é implementar a classe Retangulo:

```
package p2.exemplos;

public class Retangulo implements FiguraGeometricaPlana {
   private double lado1;
```

```
    private double lado2;

    public Retangulo(double lado1, double lado2) {
        this.lado1 = lado1;
        this.lado2 = lado2;
    }

    public Retangulo() {
        lado1 = 1.0;
        lado2 = 2.0;
    }

    // este método é idêntico ao método compareTo de circunferencia!
    @Override
    public int compareTo(FiguraGeometricaPlana outra) {
        return ((int)this.getArea() - (int)outro.getArea());
    }

    public void desenha() {
        System.out.println("Desenhando um retangulo.");
    }

    public double getArea() {
        return getLado1() * getLado2();
    }

    public double getPerimetro() {
        return 2 * getLado1() + 2 * getLado2();
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Retangulo) {
            return false;
        }

        Retangulo r = (Retangulo) obj;
```

```
      if (getLado1() == r.getLado1() && getLado2() == r.getLado2())
         return true;
      if (getLado1() == r.getLado2() && getLado2() == r.getLado1())
         return true;

      return false;
   }

   @Override
   public String toString() {
      return "Retangulo de lados " + getLado1() + " e " + getLado2() +
"."; 
   }

   /**
    * @return the lado1
    */
   public double getLado1() {
      return lado1;
   }

   /**
    * @param lado1
    *            the lado1 to set
    */
   public void setLado1(double lado1) {
      this.lado1 = lado1;
   }

   /**
    * @return the lado2
    */
   public double getLado2() {
      return lado2;
   }
```

```
    /**
     * @param lado2
     *            the lado2 to set
     */
    public void setLado2(double lado2) {
        this.lado2 = lado2;
    }
}
```

- Agora vamos mudar o programa que desenha figuras! Agora queremos que ele desenhe um retângulo e não mais uma circunferência.

```
package p2.exemplos;

public class DesenhaFiguras {

    public static void desenha( FiguraGeometricaPlana figura ) {
        figura.desenha();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        FiguraGeometricaPlana figura = new Retangulo();
        desenha(figura);
    }

}
```

- Mudamos completamente o comportamento do programa, trocando apenas a palavra Circunferencia pela palavra Retangulo!!!!
- Por que essa mudança foi tão simples!
  - As interfaces definem novos tipos
  - As interfaces também participam de uma hierarquia de tipos
  - Definimos boas interfaces (definimos bem o vocabulário do problema)

- Implementamos a interface
- Usamos o tipo (interface) ao declarar objetos e não a classe que implementa a interface
- Escrevemos chamadas polimórficas

Um repositório de figuras geométricas

- Agora queremos criar uma abstração que é um repositório de figuras geométricas
- Deve ser possível adicionar e remover figuras neste repositório
- Deve ser possível desenhar todas as figuras de uma só vez
- Vamos ao código?
  - Como esta classe deve ser chamada?
  - Que atributos ela deve ter?
  - Que métodos queremos escrever?

```
package p2.exemplos;

import java.util.*;

/**
 * Um repositorio de figuras geometricas.
 *
 * @author Raquel Lopes; Livia Campos
 *
 */

public class RepositorioDeFiguras {

   private List<FiguraGeometricaPlana> repositorio;

   /**
    * Cria um repositorio vazio de figuras geometricas.
    */

   public RepositorioDeFiguras() {
```

```
        repositorio = new ArrayList<FiguraGeometricaPlana>();

    }

    /**
     * Adiciona uma figura no repositorio.
     * @param figura
     *            A figura a ser adicionada.
     */

    public void adicionaFigura(FiguraGeometricaPlana figura) {
        repositorio.add(figura);
    }

    /**
     * Remove do repositorio a figura geometrica que esta associada a posicao
     * especificada.
     *
     * @param posicao
     *            A posicao da figura geometrica a ser removida do
     *            repositorio.
     * @return A figura geometrica removida
     * @throws Exception no caso da posicao ser invalida

     */

    public FiguraGeometricaPlana removeFigura(int posicao) throws Exception{
        return repositorio.remove(posicao);
    }


    /**
     * Se existir, desenha a figura geometrica associada a posicao
     * especificada.
     *
     * @param posicao
```

```
     *            A posicao da figura geometrica a ser desenhada
     */

    public void desenha(int posicao) {
        if (posicao >= 0 && posicao <repositorio.size())
            repositorio.get(posicao).desenha();
    }

    /**
     * Desenha todas as figuras geometricas presentes no repositorio.
     */
    public void desenhaTodasAsFiguras() {
        Iterator<FiguraGeometricaPlana> it = repositorio.iterator();
        while (it.hasNext()) {
            it.next().desenha();
        }
    }
}
```

- Note que em nenhum momento acoplamos o repositório à implementação de qualquer tipo específico de figura geométrica
  - O repositório é capaz de gerenciar qualquer figura geométrica: as que já existem… e todas as que estão por vir!
- Quer ver?
- Vamos agora escrever uma figura geométrica que é um pentágono

```
package p2.exemplos;

/**
 * Um objeto pentágono regular é considerado uma figura geométrica
 * plana com cinco lados iguais.
 *
 * @author Raquel Lopes
 *
 */
```

```java
public class PentagonoRegular implements FiguraGeometricaPlana {

    public final int NUM_LADOS = 5;
    private int lado;

    public PentagonoRegular(int lado) {
        this.lado = lado;
    }

    public int getLado() {
        return lado;
    }

    public void setLado(int lado) {
        this.lado = lado;
    }

    @Override
    public int compareTo(FiguraGeometricaPlana outro) {

        return (int) (this.getArea() - outro.getArea());
    }

    @Override
    public double getArea() {
        return (NUM_LADOS * lado * lado / (NUM_LADOS - 1)) * 1
                / Math.tan(Math.PI / NUM_LADOS);
    }

    @Override
    public double getPerimetro() {
        return NUM_LADOS * lado;
    }
```

```
    @Override
    public void desenha() {
        System.out.println("Desenhando um pentagono.");
    }

}
```

- O que precisa mudar na classe RepositorioDeFiguras para que a nova figura do tipo Pentagono também possa ser armazenada no repositório?
  - Absolutamente NADA!
- Temos mais flexibilidade quando usamos interfaces do que quando usamos herança simples
- Poderíamos pensar em outra hierarquia de classes, composta por pratos decorativos planos? Os pratos devem ter um formato de uma figura geométrica plana, mas eles não são figuras geométricas puras, pois têm uma arte associada, um material, uma cor...
- Vejamos como seria o código de um prato decorativo com formato de pentágono:

```
/**
 * Um prato decorativo que tem o formato de uma figura geométrica que é um
 * pentágono regular. Este prato tem uma arte e é usado para decoração.
 *
 * @author Raquel Lopes
 *
 */
public class PratoDecorativoGeometrico extends PratoDecorativo implements
FiguraGeometricaPlana {

    private FiguraGeometricaPlana prato;

    public PratoDecorativoGeometrico(FiguraGeometricaPlana formato) {
        prato = formato;
    }

    @Override
    public int compareTo(FiguraGeometricaPlana outroPrato) {
```

```
        return getFiguraGeometricaDoPrato().compareTo(
                outroPrato.getFiguraGeometricaDoPrato());
    }

    public FiguraGeometricaPlana getFiguraGeometricaDoPrato() {
        return prato;
    }

    @Override
    public double getArea() {
        return prato.getArea();
    }

    @Override
    public double getPerimetro() {
        return prato.getPerimetro();
    }

    @Override
    public void desenha() {
        System.out.println("Desenha um prato decorativo pentagonal");
    }

    public void mudaFormatoDoPrato(FiguraGeometricaPlana novoFormato) {
        prato = novoFormato;
    }
}
```

- Poderíamos usar o repositório de figuras para armazenar pratos decorativos?
  - SIM!
- Mas logicamente, pratos decorativos fazem parte de outra hierarquia de classes. E podemos fazer uso de polimorfismo entre hierarquias de classes diferentes, o que traz muito mais flexibilidade.
- Note que usamos composição, herança, interfaces e polimorfismo para atingir um design flexível, obter reuso e fácil de entender
  - Um PratoDecorativoGeometrico é sempre um PratoDecorativo, não importa o que aconteça. Não

pretendemos que ele mude de humor ou passe a ser um vaso, ou um prato de verdade, posto à mesa. Este é um caso típico em que herança pode ser usado. Todos os pratos decorativos compartilham "responsabilidades" em comum, relacionadas à arte da decoração do prato. Mas cada prato pode ter seu próprio formato.

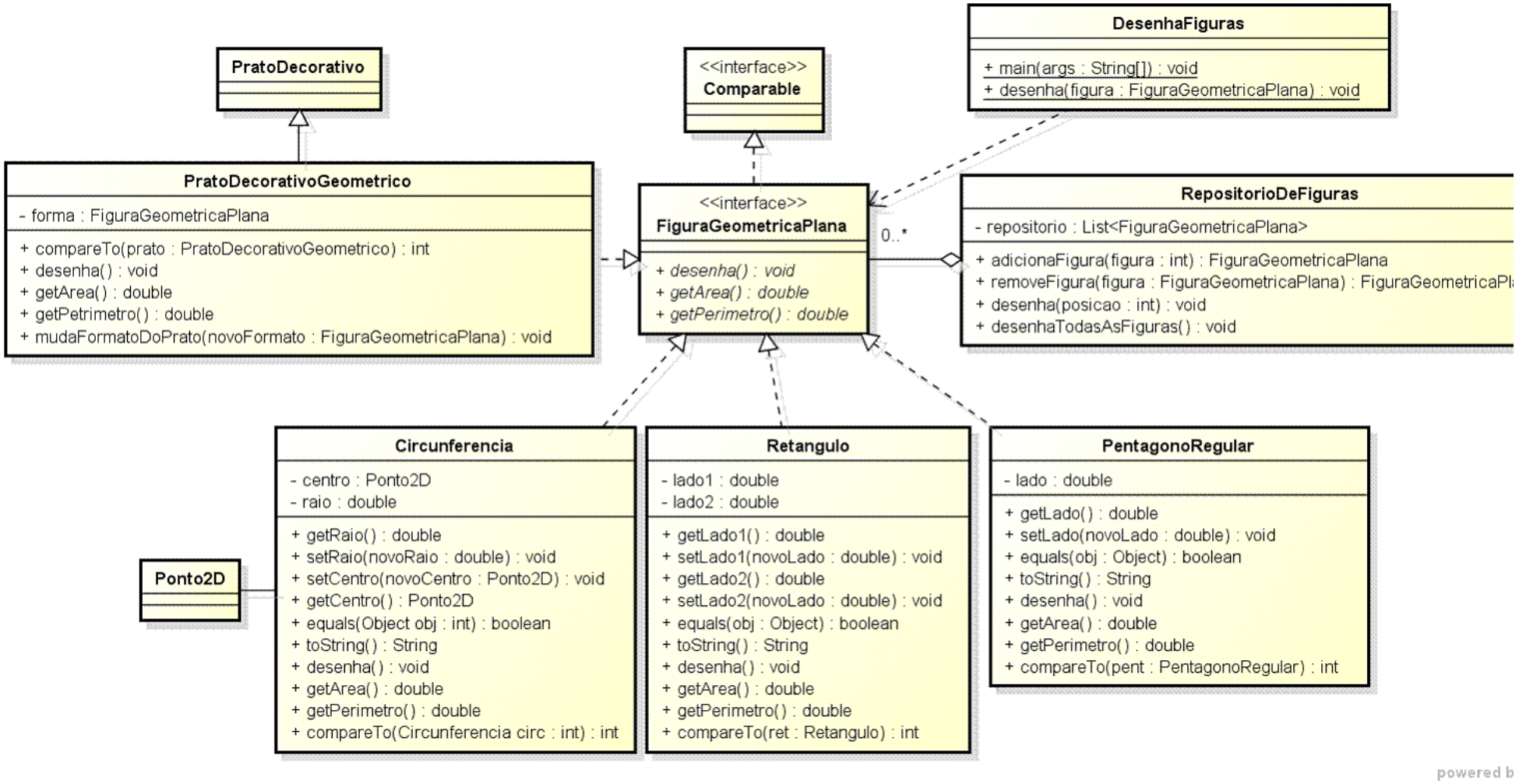
i) O formato do prato decorativo pode mudar. Ele pode passar por uma reforma... Então se o prato tivesse que estender de uma figura geométrica, ele teria que se manter no mesmo formato, uma vez que associação entre o prato e a figura que ele estende é estática e única, realizada quando o prato é criado. Seria estranho também o fato de dizer que um prato decorativo é um tipo especial de figura geométrica. Na verdade, ele pode se comportar como uma, já que tem um formato de uma figura geométrica. Então pode ter área e um desenho associados. Além disso, se prato decorativo geométrico herdasse de figura geométrica, ele não poderia herdar também de PratoDecorativo, uma vez que não existe herança múltipla em Java

ii) Usamos composição para reusar a implementação de uma figura geométrica que já existe, mas usamos a interface como uma forma de desacoplar o PratoDecorativoGeometrico a alguma implementação particular de figura geométrica. Por exemplo, poderíamos criar um prato decorativo passando pra ele um pentágono regular e assim herdar toda a implementação de pentágono regular. Ou poderíamos passar uma Circunferencia ou Retangulo ou qualquer outra implementação de FiguraGeometricaPlana que venha existir.

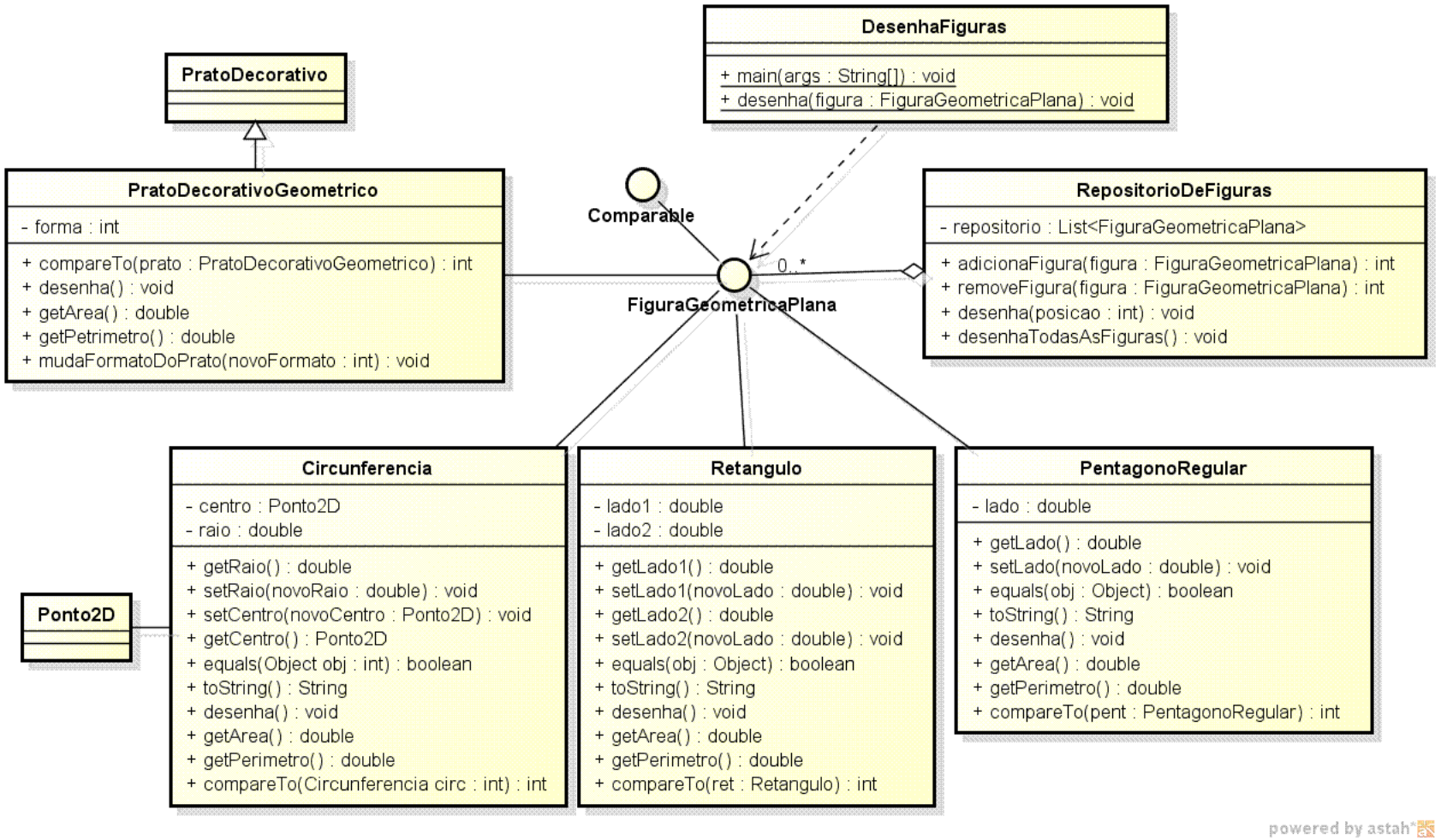
- O repositório de figuras é capaz de desenhar qualquer prato decorativo geométrico, uma vez que esses pratos se comportam como figuras geométricas. Assim, temos polimorfismo para classes que pertencem a hierarquias de tipos diferentes, sem deixar o código complexo e sem problemas na manutenção.
- Quanto mais figuras geométricas eu tiver, mais possibilidades de formatos de pratos decorativos teremos, sem precisar modificar uma linha de código da classe PratoDecorativoGeometrico. Reusamos código, não temos problemas de manutenção ao criar novos formatos de figuras.
- O que propiciou tudo isso? **Interfaces + Composição + polimorfismo**.

Representação em UML

- Em UML, interfaces são representadas de duas formas
  - Com definição completa, como “classe” usando o estereótipo <<interface>>
  - De forma resumida, usando uma notação gráfica específica (bola)
- Exemplo: nosso último programa



- Exemplo: nosso último programa mas usando a forma abreviada de interface



Qual é a diferença entre uma classe abstrata e uma interface?

- Uma classe abstrata pode ter implementação de alguns métodos e a interface não pode
- Uma classe abstrata também define uma interface (seus métodos públicos)
- Para reusar o contrato de uma classe abstrata temos que estendê-la e implementar os métodos abstratos e/ou sobrescrever os métodos que devem apresentar comportamento diferente do herdado
  - Isto causa uma relação "é um" entre a classe abstrata e a classe derivada dela
  - Muitas vezes esta relação é muito forte para ser usada!
- Para reusar o contrato de uma interface temos que implementar os métodos que ela define

- Isto causa um relacionamento “é como um” entre a interface e a classe que a implementa
- A nova classe age como um …
  i) Raquel age como uma mãe…
  ii)Raquel age como uma filha…
  iii) Raquel age como uma professora…
  iv) Raquel age como uma aluna… (na aula de francês!)
  v)Raquel pode acumular vários papéis ao mesmo tempo

- Peculiaridade de Java:
  - Em se tratando da hierarquia de classes apenas herança simples é permitida
  - Em se tratando da hierarquia de tipos abstratos (interfaces) herança múltipla é permitida
    i) Uma interface pode estender várias interfaces simultaneamente ou
    ii)Uma interface pode herdar de várias interfaces simultaneamente
    iii) Isto é simples porque não há implementação. Assim, se dois métodos tiverem a mesma assinatura, apenas uma implementação será oferecida
  - Como sugerido no exemplo dos papéis acumulados por Raquel, uma classe pode implementar várias interfaces simultaneamente
  - Vejamos o código disso!

```
package p2.exemplos;

public interface MaeIF {

   public void manda();

}
```

```
package p2.exemplos;

public interface FilhaIF {

   public void obedece();

}
```

```java
package p2.exemplos;

public interface ProfessoraIF {

   public void daAula();
   public void preparaAula();

}
```

```java
package p2.exemplos;

public interface AlunaIF {

   public void estudaParaProva();
   public void participaDeAula();
   public void fazProva();

}
```

```java
package p2.exemplos;

public class MulherOcupada extends Pessoa implements MaeIF, FilhaIF,
ProfessoraIF,
      AlunaIF {

   public MulherOcupada(String nome) {
      super(nome);
   }

   public void manda() {
      System.out.println("Mandando como mae!");
   }
```

```
    public void obedece() {
        System.out.println("Obedecendo como filha!");
    }

    public void daAula() {
        System.out.println("Professora dando aula.");
    }

    public void preparaAula() {
        System.out.println("Professora preparando aula.");
    }

    public void estudaParaProva() {
        System.out.println("Aluna estuda para a prova!");
    }

    public void participaDeAula() {
        System.out.println("Aluna participa de aula.");
    }

    public void fazProva() {
        System.out.println("Aluna fazendo prova.");
    }

}
```

```
package p2.exemplos;

public class DiaDaMulherOcupada {

    private static void manda(MaeIF mae) {
        mae.manda();
    }

    private static void obedece(FilhaIF filha) {
        filha.obedece();
```

```
    }

    private static void preparaAula(ProfessoraIF professora) {
        professora.preparaAula();
    }

    private static void estudaParaProva(AlunaIF aluna) {
        aluna.estudaParaProva();
    }

    public static void main(String[] args) {
        MulherOcupada raquel = new MulherOcupada("Raquel");
        manda(raquel); // trata raquel como uma MaeIF
        obedece(raquel); // trata raquel como uma FilhaIF
        preparaAula(raquel); // trata raquel como uma ProfessoraIF
        estudaParaProva(raquel); // trata raquel como uma AlunaIF
    }

}
```

- Em situações como essa, uma flexibilidade que as interfaces nos oferecem é a de poder fazer o *upcast* para mais de um tipo base
  - No exemplo acima fizemos o *upcast* do mesmo objeto para quatro diferentes tipos base!
- Como exemplo, podemos ter uma interface que estende várias outras
  - No código acima, poderíamos ter criado a interface da mulher ocupada que estende MaeIF, FilhaIF, ProfessoraIF e AlunaIF
  - Depois implementaríamos apenas esta interface. Também teria dado certo. Tente isto em casa!

Princípio importante de projeto orientado a objetos

Program to an interface, not to an

implementation!

Erich Gamma

Grande parte deste material que segue foi retirado da entrevista concedida por Erich Gamma a Tim Bernners (aqui).

- O que você acha que isto significa? Vamos ver?
- Quando reusamos estabelecemos dependências
- Usar interfaces faz você depender de interfaces e não de implementações, o que é uma dependência saudável
- Mas o que significa usar uma interface?
  - Significa declarar a variável como sendo do tipo da interface, mas ao fazer o new, obviamente você irá usar uma implementação da interface
- Uma interface define a colaboração entre objetos
  - Uma interface não tem detalhes de execução/implementação
  - Uma interface define o "vocabulário" da colaboração
  - Uma vez que você entende as interfaces, você entende mais do sistema porque você compreende o vocabulário do problema
  - Vocabulário = abstrações + métodos (mensagens)
- Toda classe faz parte de uma hierarquia de tipos
  - A idéia é que quanto mais alto estivermos na hierarquia, mais abstrato é o tipo (até chegarmos nas interfaces!)
  - Quanto mais descemos na hierarquia, mais concretas vão ficando as classes, associadas a implementações
  - Esta regra diz que você deve declarar seus objetos (suas referências, na realidade!) como sendo do nível mais alto possível da hierarquia. Vá até o nível mais alto onde você possa chegar!!!
  - Assim você estará desacoplado de implementações específicas. Novas implementações deste mesmo tipo abstrato podem ser facilmente incorporadas em seu programa
- Apesar de tudo que foi dito, cuidado para não sair criando interfaces sem necessidade!

  - Refactoring!
- Design patterns interessantes de serem estudados neste momento: Strategy e Factory Method!

HP disciplina - *Programa*