CONCEITOS BÁSICOS DE ORIENTAÇÃO A OBJETOS

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

Escola de Artes, Ciências e Humanidades | EACH | USP

O QUE É UM OBJETO?

Definição (apesar de não existir consenso)
Objeto é um componente de software que contém propriedades (estado) e os métodos (comportamento) necessários para tornar um tipo de dado útil.

- Um objeto é um conjunto **estado** + **comportamento**
- Estado — os dados que estão contidos no objeto
  - armazenado nos **campos** do objeto
- Comportamento — as ações disponibilizadas pelo objeto
  - Providas por **métodos**
    - Métodos são o jeito OO de dizer funções
    - invocar (*to invoke* um método = chamar uma função

CLASSES

- Todo objeto tem uma **classe**
  - uma classe define métodos e campos
  - métodos e campos são chamados de **membros** da classe
- Classes definem tanto um **tipo** quando uma **implementação**
  - Tipo $\approx$ o quê o objeto e onde ele pode ser usado
  - Implementação $\approx$ como o objeto faz as coisas
- Os métodos de uma classe definem sua **Application Programming Interface (API)**
  - Define como os usuários interagem com as instâncias

EXEMPLO DE CLASSE

```java
public class Complex {
  private final double re;
  private final double im;

  public Complex(double real, double imag) {   // Construtor
    re = real;
    im = imag;
  }

  public double realPart() { return re; }      // Métodos Acessores
  public double imaginaryPart() { return im; }

  public Complex plus(Complex b) {
    Complex a = this;
    double real = a.re + b.re;
    double imag = a.im + b.im;
    return new Complex(real, imag);
  }

  public Complex times(Complex b) {
    Complex a = this;
    double real = a.re * b.re - a.im * b.im;
    double imag = a.re * b.im + a.im * b.re;
    return new Complex(real, imag);
  }
}
```

EXEMPLO DE USO

```
public class ComplexUser {

  public static void main(String args []) {
    Complex c1 = new Complex(-1.0, 0.0);
    Complex c2 = new Complex( 0.0, 1.0);

    Complex e = c1.plus(c2);
    System.out.printf("Sum:     %.1f + %.1fi%n",
                      e.realPart(), e.imaginaryPart());

    e = c1.times(c2);
    System.out.printf("Product: %.1f + %.1fi%n",
                      e.realPart(), e.imaginaryPart());
  }
}
```

```
Sum:     -1.0 + 1.0i
Product: 0.0 + -1.0i
```

INTERFACES E IMPLEMENTAÇÕES

- Múltiplas implementações de uma API podem coexistir
  - múltiplas classes podem implementar a mesma API
- Em Java, uma API é definida pela sua *classe* ou *interface*
  - Classes fornecem uma API e uma implementação
  - Intertaces fornecem *apenas* uma API
  - Uma classe pode implementar múltiplas interfaces

UMA INTERFACE PARA NOSSA IMPLEMENTAÇÃO

```
public interface Complex {
  // sem construtores, campos ou implementações!

  double realPart();
  double imaginaryPart();

  Complex plus(Complex c);
  Complex minus(Complex c);
  Complex times(Complex c);
  Complex dividedBy(Complex c);
```

Uma interface define, mas não implementa, uma API!

MODIFICAÇÕES NA CLASSE PARA USAR A INTERFACE

```
public final class OrdinaryComplex implements Complex {
  private final double re;
  private final double im;

  private OrdinaryComplex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart() { return re; }
  public double imaginaryPart() { return im; }

  public Complex plus(Complex c) {
    return new OrdinaryComplex(re + c.realPart(),
                               im + c.imaginaryPart());
  }
}
```

MODIFICAÇÕES NO CLIENTE PARA USAR A INTERFACE

```
public class ComplexUser {

  public static void main(String args []) {
    Complex c = new OrdinaryComplex(-1, 0);
    Complex d = new OrdinaryComplex( 0, 1);

    Complex e = c.plus(d);
    System.out.printf("Sum:     %.1f + %.1fi%n",
                      e.realPart(), e.imaginaryPart());

    e = c.times(d);
    System.out.printf("Product: %.1f + %.1fi%n",
                      e.realPart(), e.imaginaryPart());
  }
}
```

```
Sum:     -1.0 + 1.0i
Product: 0.0 + -1.0i
```

INTERFACES PERMITEM MÚLTIPLAS IMPLEMENTAÇÕES

```
public final class PolarComplex implements Complex {
  final double r; // Representação diferente!
  final double theta;

  private PolarComplex(double r, double theta) {
    this.r = r;
    this.theta = theta;
  }

  public double realPart() { return r * Math.cos(theta); }
  public double imaginaryPart() { return r * Math.sin(theta); }

  public Complex plus(Complex c) { ... } // Implementação diferente!
  public Complex times(Complex c) {
    return new PolarComplex(r*c.r(), theta + c.theta());
  }
}
```

A INTERFACE DESACOPLA CLIENTE DA IMPLEMENTAÇÃO

```
public class ComplexUser {

  public static void main(String args []) {
    Complex c = new PolarComplex(1, Math.PI);
    Complex d = new PolarComplex(1, Math.PI/2);

    Complex e = c.plus(d);
    System.out.printf("Sum:     %.1f + %.1fi%n",
                      e.realPart(), e.imaginaryPart());

    e = c.times(d);
    System.out.printf("Product: %.1f + %.1fi%n",
                      e.realPart(), e.imaginaryPart());
  }
}
```

```
Sum:     -1.0 + 1.0i
Product: 0.0 + -1.0i
```

POR QUE TER MÚLTIPLAS IMPLEMENTAÇÕES?

- **Desempenhos** diferentes
  - escolha a implementação que funcionar melhor pro seu uso
- **Comportamentos** diferentes
  - escolha a implementação que faz o que você quer
  - o comportamento *precisa* respeitar a especificação da interface (“contrato”)
- Geralmente, *tanto* **desempenho quanto comportamento** variam
  - compromisso entre funcionalidade e desempenho
  - ex: `HashSet`, `LinkedHashSet`, `TreeSet`

PREFIRA USAR O TIPO DE INTERFACES AO DE CLASSES

- Use os tipos de interfaces para parâmetros e variáveis, a não ser que uma implementação simples seja suficiente
  - permite trocar as implementações
  - evita dependência em detalhes da implementação
- De vez em quando usar uma única implementação é suficiente
  - nesse caso, escreva uma classe e use-a

Faça isso: `List<Integer> lista = new ArrayList<>();`

Não isso: `ArrayList<Integer> lista = new ArrayList<>();`

... mas não é preciso exagerar

PERGUNTA

```java
interface Animal {
  void vocalize();
}

class Dog implements Animal {
  public void vocalize() { System.out.println("Woof!"); }
}

class Cow implements Animal {
  public void vocalize() { moo(); }
  public void moo() {System.out.println("Moo!"); }
}
```

O que o código abaixo faz?

1. `Animal a = new Animal(); a.vocalize();`
2. `Dog b = new Dog(); b.vocalize();`
3. `Animal c = new Cow(); c.vocalize();`
4. `Animal d = new Cow(); d.moo();`

Ocultação de informação (aka encapsulamento)

- O fator mais importante que distingue um módulo bem projetado de um mal projetado é o tanto que ele consegue esconder os seus detalhes dos dados e da implementação interna dos outros módulos
- Código bem projetado esconde *todos* os detalhes de implementação:
  - separa claramente a API de sua implementação;
  - módulos se comunicam *apenas* por suas APIs;
  - um não sabe como o outro funciona internamente.
- Princípio fundamental de projeto de software

BENEFÍCIOS DE OCULTAR A INFORMAÇÃO

- **Desacopla** as classes que compõem o sistema
  - permite que sejam desenvolvidos, testados, otimizados, usados, entendidos e modificados de forma isolada
- **Acelera o desenvolvimento de sistemas**
  - classes podem ser desenvolvidas em paralelo
- **Facilita a manutenção**
  - classes podem ser entendidas mais rapidamente e depuradas sem medo de afetar outros módulos
- **Permite ajustes de desempenho mais efetivos**
  - classes-chave podem ser otimizadas de forma isolada
- **Aumenta o reuso do software**
  - classes fracamente acopladas permitem reutilização em outros contextos

OCULTAÇÃO DE INFORMAÇÃO COM INTERFACES

- Declare variáveis usando tipos definidos pela interface
- Faça o cliente usar apenas os métodos da interface
- Campos e métodos específico da implementação não devem ficar acessíveis pelo código cliente
- Mas isso nos ajuda só até certo ponto
  - o cliente pode acessar os membros que não fazem parte da interface diretamente
  - ou seja, isso só garante uma ocultação **voluntária** da informação

Modificadores de acesso para os membros

private acessíveis apenas dentro da classe que a declara

default acessíveis apenas por uma classe no pacote onde o membro foi declarado

- é a permissão de acesso padrão, quando nada é especificado

protected acessíveis de subclasses da classe que os declara (e dentro do pacote)

public acessíveis de qualquer classe

MODIFICADORES DE ACESSO

Modificador	Classe	Pacote	Subclasse	Mundo
`public`	Sim	Sim	Sim	Sim
`protected`	Sim	Sim	Sim	Não
(sem modificador)	Sim	Sim	Não	Não
`private`	Sim	Não	Não	Não

Tabela 1: Níveis de acesso

MÉTODOS E VARIÁVEIS DE CLASSE

```
public class Bicicleta {
    private int cadência;   private int marcha;
    private int velocidade;
    private int id;
    private static int númeroDeBicicletas = 0;

    public Bicicleta(int cadênciaInicial, int velocidadeInicial,
                                          int marchaInicial){
        marcha = marchaInicial;
        cadência = cadênciaInicial;
        velocidade = velocidadeInicial;

        // incrementa o número de bicicletas
        // e o atribui como identificador
        id = ++númeroDeBicicletas;
    }

    public int getID() {
        return id;
    }

    public static int getNúmeroDeBicicletas() {
        return númeroDeBicicletas;
    }
}
```

BOAS PRÁTICAS PARA OCULTAR INFORMAÇÃO

- Projete sua API com cuidado
- Forneça *apenas* as funcionalidades necessárias para os clientes
  - *todos* os outros membros devem ser `private`
- Use o modificador de acesso mais restritivo possível
- Você sempre pode transformar um membro que era `private` em algo **public** mais tarde sem quebrar os clientes, mas não o contrário!

BIBLIOGRAFIA

- The Java™ Tutorials
  `https://docs.oracle.com/javase/tutorial/java/concepts/index.html`
- Effective Java, Items 15 e 16 e notas de aula de Josh Bloch e Charlie Garrod