Introdução a Programação – IF669
http://www.cin.ufpe.br/~if669

Interfaces

AULA 14

Ricardo Massa F. Lima
rmfl@cin.ufpe.br

Sérgio C. B. Soares
scbs@cin.ufpe.br

Centro de Informática
Universidade Federal de Pernambuco
CIn.ufpe.br

Encapsulamento e *Information Hiding*

- A habilidade em "esconder" de forma segura dados e métodos de uma classe dentro da "cápsula"da classe, impedindo acesso de usuários não confiáveis é conhecida como *information hiding*

mas...por que estamos interessados em fazer isso?

Por que ecapsulamento com *information hiding* é útil?

- Esconder detalhes de implementação
  - evita que outros programadores façam uso dessas informações com algum propósito
  - torna possível modificar a implementação com a segurança de que não afetará o código que utiliza a classe

Por que ecapsulamento com *information hiding* é útil?

- Protege a classe de interferências externas indesejáveis (sejam elas acidentais ou propositais)
  - A classe contém conjunto de campos interdependentes, que devem ser mantidos em um estado consistente
  - Se for permitido a algum usuário externo (ou você mesmo) modificar um campo sem modificar campos relacionados a ele, a classe ficará em um estado inconsistente
  - Se, por outro lado, o acesso é feito via um método, há mais chances de que o estado será mudado de maneira consistente

Por que ecapsulamento com *information hiding* é útil?

- Se todos os dados da classe estão escondidos e podem ser acessados apenas via métodos da classe e esses métodos foram bem testados, haverá maior garantia de que os dados serão modificados consistentemente
- Se, por outro lado, for permitido acesso externo, o número de possibilidades a serem testadas torna-se não gerenciável.

Por que ecapsulamento com *information hiding* é útil?

- Se atributos públicos são acessados diretamente por outras classes forem modificados, as classes que os acessam diretamente serão afetadas
  - Acessando os mesmos com métodos pode evitar esse impacto
- Se um determinado método foi definido apenas para uso interno da classe, esconder esse método evita que usuários da classe tentem usá-lo

Por que ecapsulamento com *information hiding* é útil?

Menos nobre, mas... se um campo ou método de uma classe for visível, você terá que documentá-lo

economize seu tempo e esforço!

esconda os campos e métodos ao máximo!

Interfaces

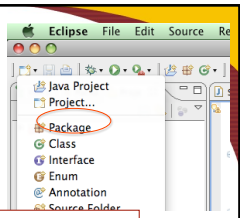
- Através do encapsulamento, os atributos e a implementação dos métodos de uma certa classe não são visíveis ao usuário da classe
- Conhecendo-se apenas a interface de uma classe, podemos utilizar seus objetos sem conhecer detalhes de implementação
- Uma interface inclui os métodos disponíveis e suas respectivas assinaturas
- Além disto, existem casos, onde existe a necessidade de se ter uma classe mas não queremos implementá-la
  - pode-se terceirizar a implementação, fornecendo como especificação a interface desejada.

Interfaces - Exemplo

- Implementar um zoológico virtual com vários tipos de animais
- Você gostaria de enviar as seguintes mensagens a cada animal:
  - `nasca()`
  - `passeie()`
  - `durma()`
  - `peso()`
- Vamos pedir ajuda a programadores especialistas em cada tipo de animal



Interfaces - Exemplo




```
public interface Animal {
        void nasca();
        void passeie();
        void durma();
        double peso();
    }
```

Interfaces - Exemplo

- O programador que for implementar o morcego terá que dizer explicitamente que vai usar a interface Animal
  - palavra chave `implements`

```
public class Morcego implements Animal{

    public void nasca(){
        System.out.println("Nasce um lindo morcego");
    }

    public void passeie(){
        System.out.println("Voa de um lado para o outro");
    }

    public void durma(){
        System.out.println("Dorme de cabeça para baixo");
    }

    public double peso(){
        return 4.5;
    }
}
```

A palavra chave implements obriga o programador a escrever o código de todos os métodos na assinatura

Todos os métodos da interface devem ser públicos

```
public interface Animal {
        void nasca();
        void passeie();
        void durma();
        double peso();
    }
```

Interfaces - Exemplo

```
public class Ornitorrinco implements Animal{

    double peso;

    Ornitorrinco(double p){
        peso = p ;
    }

    public double peso(){
        return peso;
    }
    public void nasca(){
        System.out.println("Quebra o ovo para sair");
    }

    public void passeie(){
        System.out.println("Anda e nada de um lado para o outro");
    }

    public void durma(){
        System.out.println("Dorme dentro de túneis, durante o dia");
    }
}
```

```
public interface Animal {
        void nasca();
        void passeie();
        void durma();
        double peso();
    }
```

Interfaces - Exemplo

```
public class Zebra implements Animal{
    int listras;
    double peso;

    public Zebra (int l, double p){
        listras = l;    // cria uma zebra com l listras
        peso = p;       // e peso p
    }

    public void nasca(){
        System.out.println("Nasce mais uma zebra");
    }

    public void passeie(){
        System.out.println("Galopa pelo campo");
    }

    public void durma(){
        System.out.println("Dorme em pé");
    }

    public double peso(){
        return peso;
    }

    // nada impede que sejam implementados métodos adicionais
    public void contaListras(){
        System.out.println("Esta zebra tem"+listras+"listras");
    }
}
```

```
public interface Animal {
        void nasca();
        void passeie();
        void durma();
        double peso();
    }
```

Interfaces - Observação

Em cada arquivo deve existir no máximo uma classe pública!

Logo, as classes Ornitorrinco, Morcego e Zebra devem estar em arquivos separados, com os respectivos nomes

Ornitorrinco.java

Zebra.java

Morcego.java

Interfaces

Cada um dos animais, além de ser um objeto da própria classe, também é um objeto do tipo Animal

```
public class ZoologicoVirtual {

    static public void cicloDeVida(Animal animal){
        animal.nasca();
        animal.passeie();
        animal.durma();
    }

    static public void fazFuncionar(){
        Zebra z1        = new Zebra(102,99);          //cria duas zebras
        Animal z2       = (Animal) new Zebra(101,107); //sendo uma do tipo Animal
        Morcego m1      = new Morcego();
        Ornitorrinco o1 = new Ornitorrinco(25);

        cicloDeVida(z1);
        cicloDeVida(z2);
        cicloDeVida(m1);
        cicloDeVida(o1);
        }
}
```

z1.contaListras() - Válido

z2.contaListras() - Inválido

```
> ZoológicoVirtual.fazFuncionar()
Nasce mais uma zebra
Galopa pelo campo
Dorme de pé
Nasce mais uma zebra
Galopa pelo campo
Dorme de pé
Nasce um lindo morcego
Voa de um lado para o outro
Dorme de ponta cabeça
Quebra o ovo para sair
Anda e nada de um lado para o outro
Dentro de túneis, durante o dia
>
```

fazFuncionar - Refatorando

Versão Refatorada 1

```
static public void fazFuncionar(){

        Animal[] bicharada = new Animal[4];

        bicharada[0] = new Zebra(102,99);
        bicharada[1] = new Zebra(101,107);
        bicharada[2] = new Morcego();
        bicharada[3] = new Ornitorrinco(25);

        for(int i=0; i<bicharada.length; i++)
            cicloDeVida(bicharada[i]);
        }
```

Versão Refatorada 2

```
    static public void fazFuncionar(){
        Animal[]bicharada=
            {   new Zebra(102,99),   new Zebra(101,107),
                new Morcego(),       new Ornitorrinco(25)};

        for(int i=0;i<bicharada.length;i++)
            cicloDeVida(bicharada[i]);
    }
```

Implementando mais de uma interface por vez

- Considere as duas interface:

```
public interface TransportadorDePessoas {
    void entraPessoas();
    void saiPessoas();
}
```

```
public interface Voador {
    void voa();
    void aterrisa();
}
```

- Vamos implementar essas três classes

Implementando mais de uma interface por vez

```
class Ave implements Voador{

    public void voa(){
        System.out.println("Bateasasasbemforte");
    }

    public void aterrisa(){
        System.out.println("Bateasasasmaisfracoepôeospésnochão");
    }

}
```

```
class Onibus implements TransportadorDePessoas{

    public void entraPessoas(){
        System.out.println("Abreasportaseentramaspessoas");
    }

    public void saiPessoas(){
        System.out.println("Abreasportasesaemaspessoas");
    }
}
```

Implementando mais de uma interface por vez

Finalmente, podemos ver o `Aviao` que implementa as duas interfaces:

```
class Aviao implements Voador, TransportadorDePessoas{

    public void voa(){
        System.out.println("Ligaasturbinas;recolheotremdepouso");
        }

    public void aterrisa(){
        System.out.println("Abaixaotremdepousoedesce");
        }

    public void entraPessoas(){
        System.out.println("Procedimentodeembarque");
        }

    public void saiPessoas(){
        System.out.println("Procedimentodedesembarque");
        }
}
```

Mas e na aplicação bancária? Onde usar interfaces?

- Hoje a classe Banco tem um array de Conta
- E se amanhã quisermos utilizar outra estrutura de dados?
- E se quisermos depois de amanhã utilizar um banco de dados?
- Vamos desacoplar as regras de negócio do Banco de onde as contas são armazenadas

Criar uma interface de armazenamento de dados

```
public interface RepositorioContas {
  void inserir(ContaAbstrata conta);
  ContaAbstrata procurar(String numero);
  void remover(String numero);
  void atualizar(ContaAbstrata conta);
  boolean existe(String numero);
}
```

Todos os métodos são `public` e `abstract` por *default* e não se definem atributos nem construtores

Repositório: Implementações

```
public class RepositorioContasArray
        implements RepositorioContas {...}

public class RepositorioContasLista
        implements RepositorioContas {...}

public class RepositorioContasVector
        implements RepositorioContas {...}

public class RepositorioContasBDR
        implements RepositorioContas {...}
```

Banco: Parametrização

```
public class Banco {
  private RepositorioContas contas;
  public Banco(RepositorioContas rep){
    this.contas = rep;
  }
  public void cadastrar(ContaAbstrata conta){
    String numero = conta.getNumero();
    if (!contas.existe(numero)) {
      contas.inserir(conta);
    } else {
      throw new RuntimeException("Já cad...");
    }
    // ...
}
```

A estrutura para armazenamento das contas é fornecida na inicialização do banco, e pode ser trocada!

O que usar? Quando?

Classes (abstratas)

- Agrupa objetos com implementações compartilhadas
- Define novas classes através de herança (simples) de código
- Uma classe pode ter apenas uma como superclasse

Interfaces

- Agrupa objetos com implementações diferentes
- Define novas interfaces através de herança (múltipla) de assinaturas
- Uma classe pode ter várias como supertipo

Exercício

- Utilize a solução dos exercícios da última aula
  http://www.cin.ufpe.br/~if669/material/solucoes/aula13.zip
- Defina no pacote `aula14.br.ufpe.cin.dados` a interface `RepositorioContas` com os métodos
  - `inserir` – recebe uma ContaAbstrata e insere no repositório
  - `procurar` – recebe um número e retorna a conta se estiver no repositório
  - `remover` – recebe um número para remover a conta do repositório
  - `atualizar` – recebe uma ContaAbstrata para atualizar no repositório
  - `existe` – recebe um número e informa se existe uma conta com este número no repositório
- Modifique a classe `Banco` para utilizar a interface definida (receba a implementação da interface no construtor)
  - Perceba que a classe `Banco` **compila** apenas com a interface. Já a classe `Programa` precisa de uma implementação para executar.
- Defina no pacote `aula14.br.ufpe.cin.dados` classe `RepositorioContasArray` que implementa a interface `RepositorioContas`
- Altere a classe `Programa` para fazer os testes