



Interfaces



Interfaces

Introdução



- Uma Interface é um "contrato" em que as classes que a implementam se comprometem a seguir.
- Uma interface pode conter:
 - Declaração de métodos abstratos (sem implementação)
 - *As classes que implementam a interface devem implementar esses métodos*
 - Constantes estáticas
 - *As classes que implementam a interface terão acesso às constantes*

Nota: a versão 8 do Java, permite declarar métodos estáticos nas Interfaces, dentre outras funcionalidades

Interfaces

Introdução



- Uma classe que implementa uma interface deve implementar todos os métodos definidos nesta última
 - Ou ser declarada como "abstract" (a classe)
 - Contrato da classe: "Ou eu implementarei todos os métodos da interface ou eu serei declarado como abstrata"
- Interfaces não podem ser instanciadas, apenas "implementadas" por uma classe
 - De forma semelhante às classes abstratas, que não podem ser instanciadas, apenas "herdadas"



- Enquanto que uma classe abstrata diz:
 - Um conjunto de classes relacionadas (subclasses) irão obrigatoriamente implementar um ou mais métodos abstratos
- Uma Interface diz:
 - Um conjunto de classes *não relacionadas* irão implementar um ou mais métodos abstratos
 - Desta forma, interfaces podem tornar desnecessário que classes não-relacionadas, mas que possuem um ou mais métodos em comum, precisem ter uma superclasse abstrata em comum

Interfaces

Exemplo



- O código abaixo declara uma interface chamada `Desenhavel`
 - Nome do arquivo: `Desenhavel.java`

```
public interface Desenhavel {  
  
    int COR_VERMELHO = 1;  
    int COR_VERDE = 2;  
    int COR_AZUL = 3;  
  
    void setCor(int cor);  
    void desenha();  
  
}
```

Constantes estáticas,
precisam ser inicializadas

Métodos abstratos

- Note como os atributos são automaticamente setados para constantes, mesmo que não tenhamos declarados
 - São setados como *“public static final”*
- Note como os métodos são automaticamente setados para abstratos
 - São setados como *“public abstract”*

Interfaces

Exemplo



- Para indicar que uma classe implementa uma interface, usamos a palavra reservada `implements`

```
public class Cachorro extends Mamifero implements Desenhavel {  
  
    // Outros métodos ..  
  
    public void setCor(int cor) {  
        // Seta a cor do cachorro a ser pintado na tela ...  
    }  
  
    public void desenha() {  
        // Desenha um cachorro na tela ...  
    }  
  
}
```

Implementa o método setCor da interface Desenhavel

Implementa o método desenha da interface Desenhavel

Interfaces

Uso de Interfaces como Tipos



- Fora o fato de não poderem ser instanciadas, as Interfaces podem ser utilizadas como referências para objetos de classes que a implementam
 - Da mesma forma como podemos usar referências de classes abstratas

```
Cachorro cachorro = new Cachorro();  
Desenhavel desenho = cachorro;  
desenho.setCor(Desenhavel.COR_VERDE);  
desenho.desenha();
```

A variável desenho pode referenciar qualquer objeto de uma classe que implemente a interface Desenhavel

Como a variável desenho é do tipo Desenhavel, ela só pode executar os métodos dessa interface

Interfaces

Interfaces Baseadas em Outras



- Interfaces podem estender outras interfaces
 - Diferentemente das classes, uma interface pode estender mais de uma interface
 - A classe que implementá-la, deverá implementar os métodos de todas as interfaces.

```
public interface Escalavel { ... }

public interface Rotacionavel { ... }

public interface Reflectivel { ... }

public interface Transformavel extends Escalavel,
    Rotacionavel, Reflectivel { ... }

public interface ObjetoDesenho extends Desenho,
    Transformavel { ... }

public class Forma implements ObjetoDesenho { ... }
```

A classe Forma precisará implementar todos os métodos definidos em todas as interfaces

Interfaces

Uso das Interfaces



- Interfaces são muito usadas em casos em que classes não relacionadas entre si precisam compartilhar métodos e/ou constantes comuns. Exemplos:
 - Interface `Runnable` do Java permite que qualquer classe que a implemente seja usada como uma `Thread`, pois tem o método "run"

```
package java.io;  
  
public interface Runnable {  
    public abstract void run();  
}
```

Esse é o código completo da interface `Runnable`

- Interface `Closeable` indica um objeto que precisa ser fechado. Esta interface é implementada pela classe `Scan`, usada para ler do teclado

```
package java.io;  
  
import java.io.IOException;  
  
public interface Closeable extends AutoCloseable {  
    public void close() throws IOException;  
}
```

Interfaces

Uso das Interfaces II



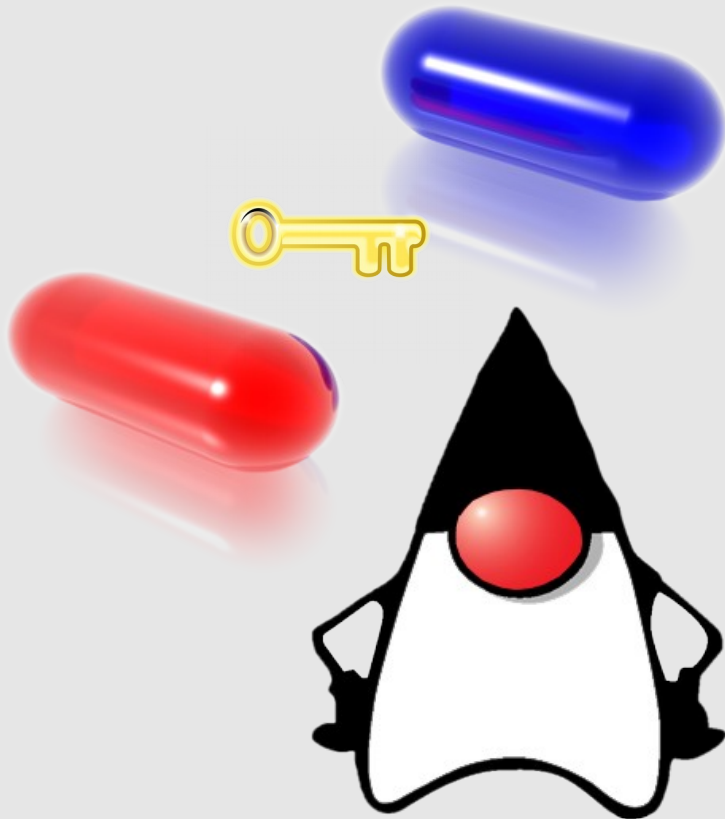
- Outra utilidade de uma interface é “marcar” uma determinada classe com uma característica. Neste caso, a interface não possui nem métodos nem constantes. Exemplos:

- interface `Serializable` do Java (que não possui nem métodos nem constantes) permite marcar qualquer classe como sendo serializável (pode ser convertido para texto e, a partir do texto, ser convertido de volta)

```
package java.io;  
  
public interface Serializable {  
}
```

- interface `Cloneable` permite marcar qualquer classe como sendo clonável (necessário para que objetos de uma classe possam ser clonados)

```
package java.lang;  
  
public interface Cloneable {  
}
```

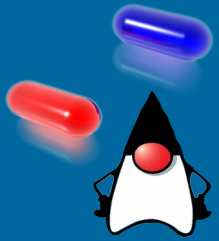


Encapsulamento



Encapsulamento

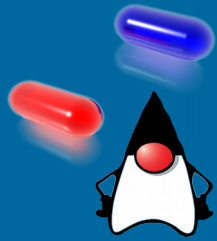
Introdução



- Encapsulamento:
 - Agrupamento de atributos e métodos em um único componente
 - Permitindo o acesso seletivo aos mesmos
 - *Acesso seletivo:*
 - *Permite o acesso*
 - *Impede o acesso (esconde)*

Encapsulamento

Exemplos na Vida Real



- Relógio

- Não precisamos saber como funciona
- Apenas como utilizá-lo



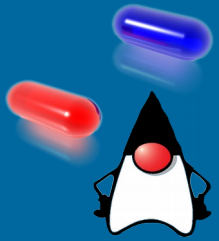
- Carro

- Não precisamos saber como funciona
- Apenas como dirigi-lo



Encapsulamento

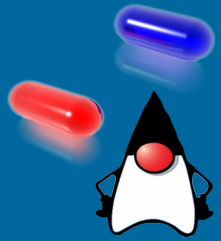
Vantagens



- Ocultamento de Informação
 - O mundo “vê” um objeto pelo que ele é capaz de fazer, e não como ele faz
 - Exemplos:
 - *Não precisamos saber como um relógio funciona internamente para usá-lo*
 - *Precisamos apenas conhecer a sua interface (API) que é*
 - *Pegar a hora atual*
 - *Setar a hora correta*
 - *Não precisamos saber como um carro funciona internamente para usá-lo*
 - *Precisamos apenas conhecer a sua interface (API) que é*
 - *Acelerar, frear, virar*
 - *Saber quanto de gasolina temos, qual a rotação do motor, etc*
 - *Não precisamos saber como a classe String funciona internamente para usarmos Strings em nossos programas*
 - *Precisamos apenas conhecer a sua interface (API) que é*
 - *Construtores*
 - *Métodos para concatenar, pesquisar*
 - *Saber o tamanho de uma string, etc*

Encapsulamento

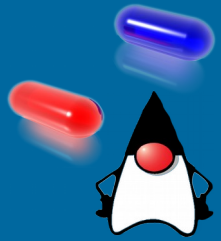
Vantagens



- Impedir o acesso a atributos e métodos internos (privados)
 - Garantindo que a classe funcione conforme planejado
 - Exemplos:
 - *Classe Pessoa*
 - *O que aconteceria se setássemos o atributo idade para um valor negativo?*
 - *Como ter certeza que o atributo CPF foi setado para um valor válido?*
 - *Como ter certeza que o atributo dataNascimento possui uma data válida?*
 - *Classe Relógio*
 - *Como ter certeza que uma data setada é válida?*

Encapsulamento

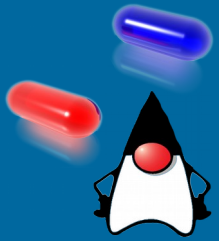
Vantagens



- Manutenção do Código
 - É possível mudar completamente como uma classe foi implementada, sem se preocupar com quem está usando a classe
 - Considerando que a implementação interna da classe foi “escondida”
 - Se mantivermos a API inicial, garantimos que todos que usam a classe antiga poderão usar a nova implementação da classe
 - Por exemplo:
 - *Você criou no passado uma lista de livros usando uma lista encadeada*
 - *Agora você resolveu mudar a implementação para usar um vetor*
 - *Como você escondeu todo o funcionamento interno da classe (e.g., a lista), você pode simplesmente*
 - *remover a lista encadeada*
 - *criar o vetor que será a nova forma de armazenamento*
 - *reimplementar os métodos públicos da classe, ou seja, sua API*
 - *adicionar, remover, buscar, etc*

Encapsulamento

Vantagens



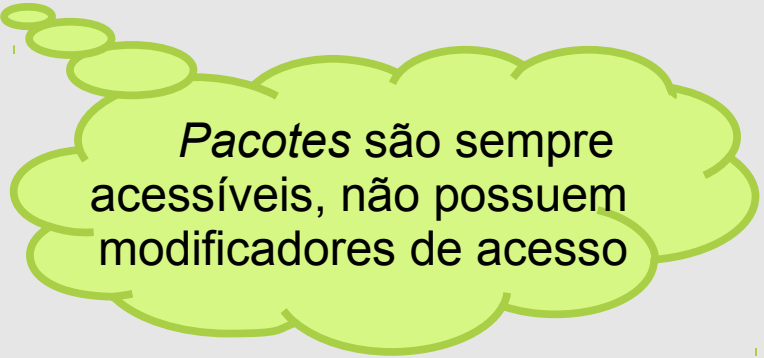
- Reusabilidade
 - Podemos utilizar qualquer classe conhecendo apenas a sua interface (atributos e métodos públicos)
 - Sem precisarmos nos preocupar como ele foi implementado

Encapsulamento

Modificadores de Acesso



- Em Java, o encapsulamento é implementado através dos modificadores de acesso
- Modificadores de acesso
 - Permitem controlar o acesso aos
 - *Atributos*
 - *Métodos*
 - *Classes*
 - *Interfaces*



Pacotes são sempre acessíveis, não possuem modificadores de acesso

A green thought bubble with a black outline, containing text about package accessibility.

Encapsulamento

Modificadores de Acesso



- Um “acesso” especifica quais as partes do código fonte que a entidade declarada poderá ser referenciada pelo seu nome
 - Exemplos:
 - *Quais atributos poderão ser acessados diretamente (pelo nome do atributo)*
 - *Quais métodos poderão ser executados diretamente*
 - *Quais classes poderão ser instanciadas pelo nome*

Encapsulamento

Modificadores de Acesso



- Java possui quatro modificadores de acesso (em ordem de restrição):
 - *Public*
 - *Protected*
 - *Package Access, Default* ou *Friendly* (padrão)
 - *Private*

Encapsulamento

Modificadores de Acesso – Exemplos



```
package br.edu.ufam.icomp.biblioteca;  
import java.util.Date;
```

```
public class Autor {  
    public String nome;  
    protected Date dataNascimento;  
    String instituicao;  
    private long cpf;
```

```
    public boolean setCPF(long cpf) {  
        if ( verificaCPF(cpf) ) {  
            this.cpf = cpf;  
            return true;  
        }  
        else return false;  
    }
```

```
    private boolean verificaCPF(long cpf) {  
        // Verifica se o CPF é válido  
        return true;  
    }  
}
```

Classe pública

Atributo público

Atributo protegido

Atributo *friendly*

Atributo privado

Método público

Executando um método privado (local)

Acessando um atributo privado (local)

Método privado

Encapsulamento

Acesso *Public*



- *Public* é o modificador de acesso mais permissível
 - Permite o acesso a partir de qualquer classe em qualquer pacote
 - *Desde que a classe que tem o atributo/método seja “observável” na classe atual, ou seja, se elas não estiverem no mesmo pacote, a primeira deve ser importada usando o import*
- Atributos com acesso *Public*:
 - São visíveis por qualquer outra classe em Java
 - Seus valores podem ser lidos e escritos diretamente
- Métodos com acesso *Public*:
 - São visíveis por qualquer outra classe em Java
 - Podem ser executados diretamente



- Classes com acesso *Public*:
 - Podem ser instanciadas dentro de qualquer outra classe,
 - Qualquer classe pode criar objetos de uma classe pública
 - Um arquivo .java só pode ter uma única classe pública, e esta deverá ter o mesmo nome do arquivo
 - Apesar de ser pouco comum, é possível que um arquivo .java tenha uma classe pública e outras classes com outros acessos, mas esta prática não é recomendada

Encapsulamento

Acesso *Public* – Exemplo



```
package br.edu.ufam.icomp.geometrico;
```

```
public class Ponto {
```

```
    public int x, y;
```

```
    public double distancia(Ponto p2) {
```

```
        return Math.sqrt(Math.pow(p2.x-this.x, 2) +  
                               Math.pow(p2.y-this.y, 2));
```

```
    }
```

```
}
```

Classe pública

Atributos públicos

Método público

Faz com que a classe Ponto seja observável

```
package br.edu.ufam.icomp;
```

```
import br.edu.ufam.icomp.geometrico.*;
```

```
public class Principal {
```

```
    public static void main(String args[]) {
```

```
        Ponto p1 = new Ponto();
```

```
        Ponto p2 = new Ponto();
```

```
        p2.x = 2;
```

```
        p2.y = 2;
```

```
        System.out.println( p1.distancia(p2) );
```

```
    }
```

```
}
```

Instanciando um objeto da classe pública Ponto

Alterando o valor de um atributo público diretamente

Executando um método público

Encapsulamento

Acesso *Protected*



- O modificador *protected* permite acesso:
 - a partir de uma subclasse da classe atual; ou
 - a partir de uma classe dentro do mesmo pacote

Encapsulamento

Acesso *Protected*



- Atributos e Métodos com acesso *Protected*:
 - São visíveis em subclasses ou classes pertencentes ao mesmo pacote
 - *No caso de atributos, seus valores poderão ser lidos e escritos diretamente*
 - *No caso de métodos, estes poderão ser executados*
 - No caso das subclasses, os atributos *protected* da superclasse serão herdados pela subclasse, ou seja, estas estarão visíveis na subclasse (como explicado na herança)
- Classes: não é possível criar classes *Protected*

Encapsulamento

Acesso *Protected* – Exemplo

```
package br.edu.ufam.icomp.geometrico;  
  
public class Ponto {  
    protected int x, y;  
}
```

```
package br.edu.ufam.icomp.geometrico;  
  
public class Ponto3D extends Ponto {  
    protected int z;  
  
    public Ponto3D(int x, int y, int z) {  
        this.x = x; this.y = y; this.z = z;  
    }  
  
    public double distancia(Ponto3D p2) {  
        return Math.sqrt(Math.pow(p2.x - this.x, 2) +  
            Math.pow(p2.y - this.y, 2) + Math.pow(p2.z - this.z, 2));  
    }  
}
```

Herdando um atributo protegido (subclasse)

Acessando um atributo protegido (subclasse)

```
package br.edu.ufam.icomp;  
import br.edu.ufam.icomp.geometrico.*;  
  
public class Principal {  
    public static void main(String args[]) {  
        Ponto3D p1 = new Ponto3D(0, 0, 0);  
        Ponto3D p2 = new Ponto3D(2, 2, 2);  
        // p2.x = 2;  
        System.out.println( p1.distancia(p2) );  
    }  
}
```

Não é mais possível acessar o atributo x diretamente, pois Principal não é subclasse de Ponto e estão em pacotes diferentes

Encapsulamento

Acesso *Package* (padrão)



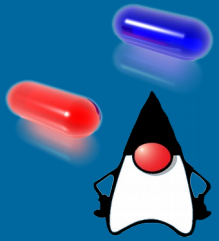
- Quando nenhum modificador é especificado, considera-se o acesso como *package*, também conhecido como *default* ou *friendly*
 - permite acesso apenas a partir de uma classe dentro do mesmo pacote
- Classes, Atributos e Métodos com acesso *Package*:
 - São visíveis em classes pertencentes ao mesmo pacote
 - *No caso de atributos, seus valores poderão ser lidos e escritos diretamente*
 - *No caso de métodos, estes poderão ser executados*
 - *No caso de classes, estas poderão ser instanciadas*
 - Exemplos: qualquer um feito nas aulas anteriores

Encapsulamento

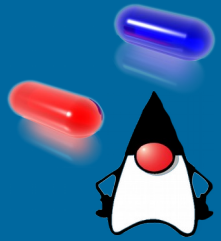
Acesso *Private*



- Private é o modificador de acesso mais restritivo
 - permite acesso apenas dentro da mesma classe
- Atributos e Métodos com acesso *Private*:
 - são estritamente controlados
 - não podem ser acessados por nenhum lugar fora da classe atual
- Classes normais não podem ser declaradas como *Private*
 - Apenas classes internas, que não serão vistas nesse curso



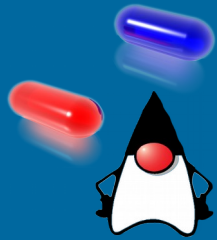
- A melhor forma de implementar encapsulamento em Java é declarando todos os atributos de uma classe como sendo *private*
 - Isso irá evitar o acesso direto aos atributos da classe, evitando que valores inválidos possam ser atribuídos a ele
 - Permite também esconder como a classe foi implementada, de forma que a mesma possa ser modificada futuramente sem muita preocupação
 - Se um atributo é privado, como seu valor poderá ser acessado e modificado?
 - *Através de métodos getters e setters*



- Métodos *getters*, retornam o valor de um atributo *private*
 - Permitindo, inclusive, retornar um valor diferente, dependendo de alguma lógica
- Métodos *setters* permitem alterar o valor de um atributo *private*
 - Permitindo, antes da alteração, que validações possam ser feitas
- Os métodos *getters* e *setters* são tão importantes e comum, que o Eclipse é capaz de gerá-los automaticamente
 - Menu *Source* → *Generate Getters and Setters ...*

Encapsulamento

Acesso *Private*: Getters e Setters



```
public class Livro {  
    private String titulo;  
    private int anoPublicacao;  
  
    public String getTitulo() {  
        if (titulo == null || titulo.equals(""))  
            return "Titulo não definido!";  
        else  
            return titulo;  
    }  
  
    public void setTitulo(String titulo) {  
        this.titulo = titulo;  
    }  
  
    public int getAnoPublicacao() {  
        return anoPublicacao;  
    }  
  
    public void setAnoPublicacao(int anoPublicacao) {  
        if (anoPublicacao > 0)  
            this.anoPublicacao = anoPublicacao;  
        else  
            this.anoPublicacao = 0;  
    }  
}
```

Atributos privados

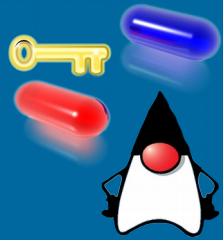
Getter para o atributo `titulo`, que retorna um valor diferente, caso este não tenha sido setado

Setter para o atributo `titulo`

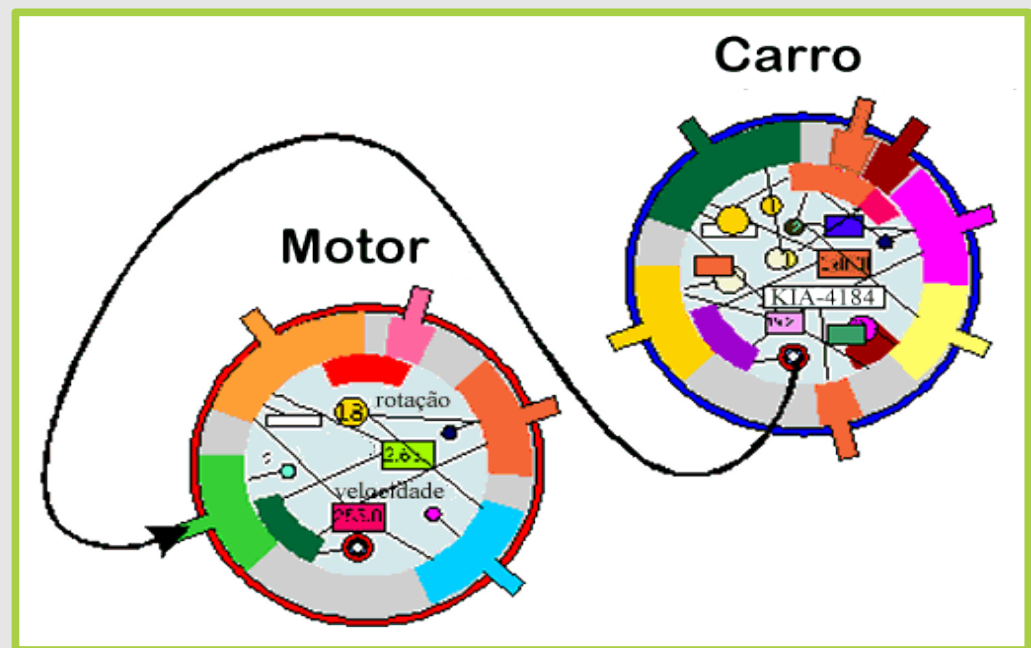
Getter para o atributo `anoPublicacao`

Setter para o atributo `anoPublicacao`, que seta um valor diferente, caso encontre um ano inválido

Encapsulamento Conclusão

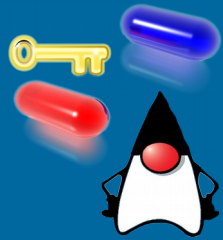


- Encapsulamento
 - Permite que uma classe se torne uma caixa preta, escondendo a implementação e permitindo o acesso através dos métodos públicos
 - Objetos de uma classe irão se comunicar com os objetos das outras classes através dos métodos públicos



Encapsulamento

Conclusão



- Resumo
 - `public`: campos serão visíveis por todos
 - `protected`: campos visíveis para as subclasses e para as classes dentro do mesmo pacote
 - sem modificador: campos acessíveis apenas pelas classes no mesmo pacote. Conhecido como "*default*" ou "*package access*"
 - `private`: esconde os membros do resto do mundo. O ideal é que todos os atributos sejam *private* e sejam acessados através de *getters* e *setters*
 - Extra:
 - *private protected*: membros visíveis apenas às subclasses

Encapsulamento

Laboratório 5

- <http://tinyurl.com/slides-tp>
 - Laboratórios
 - TP – 5oLaboratorio.pdf
- Entrega por E-Mail
 - Para: horacio.fernandes@gmail.com
 - Cópia: moyses.lima@icomp.ufam.edu.br
 - Assunto: TP: 5o Lab
- Data Limite:
 - Hoje, às 12hs
 - E-Mails recebidos após 12hs não serão considerados

