Universidade Federal do Amazonas
Instituto de Computação
Projeto de Programas
Técnicas Avançadas de Programação

Interfaces

Horácio Fernandes
horacio@icomp.ufam.edu.br

ColabWeb: bit.ly/pp-colabweb

Interfaces

- Uma Interface é um “**contrato**” em que as classes que a implementam se comprometem a seguir

- Em sua forma mais básica, uma interface **contém uma lista de métodos abstratos** (sem implementação)
  - *As classes que implementam a interface precisam implementar esses métodos*

- Entretanto, uma interface pode conter também:
  - Constantes estáticas
  - Métodos estáticos
  - Métodos *default*
    - *Possuem uma implementação padrão*
  - Métodos privados (*private*)

Usando uma Interface

- 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 declarada 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”

Interfaces × Classes Abstratas

- 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
  - Ou seja, interfaces permitem que objetos tenham comportamentos (métodos) em comum sem compartilharem uma superclasse

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** definidos como constantes, mesmo que não tenhamos declarados
    - *São definidos como "`public static final`"*
  - Note como os métodos são automaticamente definidos como abstratos
    - *São definidos como "`public abstract`"*

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`

Uso de Interfaces como Tipos

- Apesar 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 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 class Forma implements Transformavel { ... }
```

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

Uso das Interfaces

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

```
package java.lang;

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 `Scanner`, usada para ler do teclado

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

Uso das Interfaces

- Uma interface pode "**marcar**" uma determinada classe com uma característica. Neste caso, a interface não possui métodos:
  - 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 {
}
```

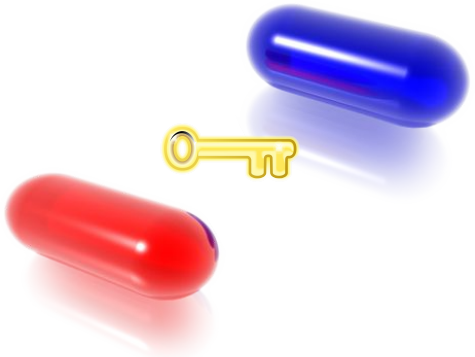
Esse é o código completo da interface `Serializable`

  - 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 {
}
```

Universidade Federal do Amazonas
Instituto de Computação
Projeto de Programas
Técnicas Avançadas de Programação



Encapsulamento

Horácio Fernandes
horacio@icomp.ufam.edu.br

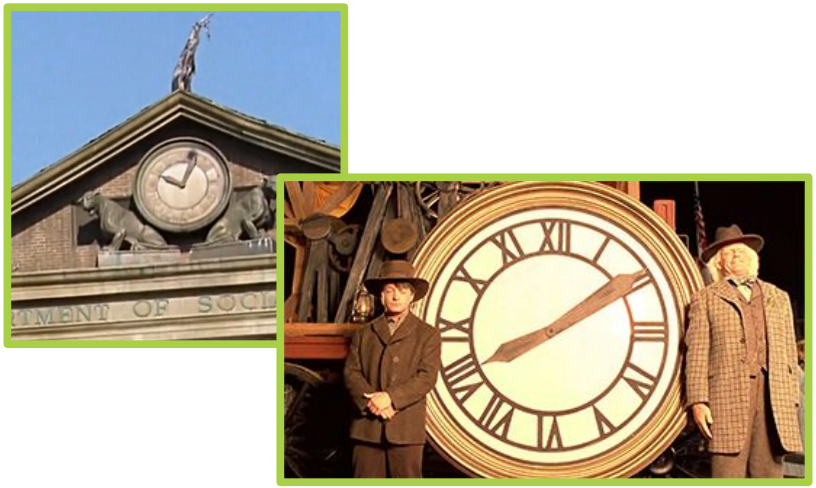
ColabWeb: bit.ly/pp-colabweb

Encapsulamento

- **Ocultação** dos detalhes internos de implementação de um objeto
- **Modularização** do sistema em blocos/módulos fechados
- **Acesso seletivo** aos atributos e métodos dos objetos
  - Permite o acesso a métodos e atributos
  - Impede o acesso (esconde)

Exemplos

- Relógio
  - Não precisamos saber como funciona
  - Apenas como utilizar



- Carro
  - Não precisamos saber como funciona
  - Apenas como dirigir

Vantagens

- Ao **ocultar detalhes** de implementação, o encapsulamento permite:
  - Ocultar Informações desnecessárias para o uso do módulo (classe)
  - Impedir o acesso a atributos e métodos internos
  - Manutenção de código
  - Reusabilidade
- Ao **modularizar** o sistema, o encapsulamento permite
  - Reusabilidade
  - Abstração
  - Associação, Composição e Agregação
- A seguir, detalharemos tais vantagens

Ocultamento de Informação

- 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:
    - *Precisamos apenas conhecer a sua interface (API): ler e definir a hora atual*
  - Não precisamos saber como um carro funciona internamente:
    - *Precisamos apenas conhecer a sua interface (API): acelerar, frear, virar, se tem gasolina*
  - Não precisamos saber como a classe `String` funciona **internamente** para usarmos strings em nossos programas
    - *Precisamos apenas conhecer a sua interface (API):*
    - *Construtores*
    - *Métodos para concatenar, pesquisar*
    - *Saber o tamanho de uma string, etc*

Controle de Acesso

- **Impedir o acesso** a atributos e métodos internos (privados)
  - Garantindo que a classe funcione conforme planejado
- Exemplos:
  - Classe Pessoa
    - *O que aconteceria se definíssemos 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?*

Manutenção de Código

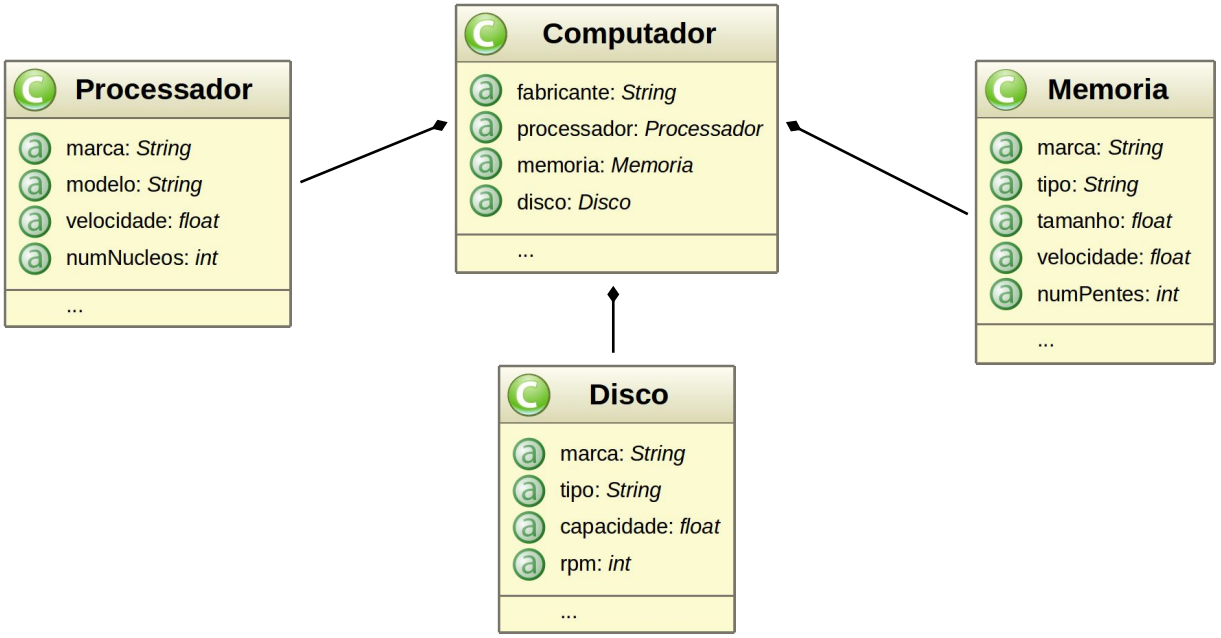
- Outra vantagem do encapsulamento é a manutenção de código
  - É possível **mudar completamente como uma classe foi implementada**, sem se preocupar com quem está usando a classe
  - Considerando que a implementação interna da classe foi “escondida”
  - Se mantivermos a API inicial, garantimos que todos 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 e re-implementar os métodos públicos da classe, ou seja, sua API adicionar, remover, buscar, etc.*

Reusabilidade

- O encapsulamento favorece a reusabilidade
- Podemos utilizar qualquer classe conhecendo apenas a sua API (atributos e métodos públicos)
  - Sem precisarmos nos preocupar como ele foi implementado
  - Por ser um “bloco” ou um “módulo”, esta classe pode ser reutilizada em outras partes do mesmo sistema ou até mesmo em outros sistemas diferentes.

Associação, Composição e Agregação

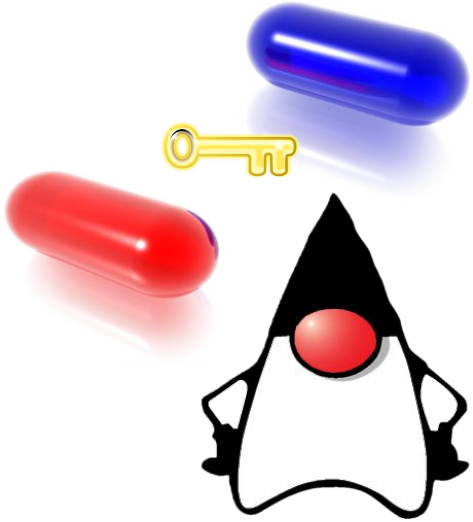
- Esta característica de modularização da Orientação a Objetos permite que a **descrição de uma classe** (módulo) possa ser feita **contendo outras classes** (módulos). Exemplo:



Associação, Composição e Agregação

- No exemplo anterior, um `Computador` é composto de outros objetos. Ele contém um `Processador`, um `Disco` e uma `Memória`
- Isso é conhecido como **associação**
- Apesar de não existir diferenças em Java, na modelagem de sistemas, existem dois subtipos de associações:
  - **Agregação**: quando uma parte pode existir por si só, sem depender do todo. Por exemplo, um carro pode conter passageiros ou não. Portanto, existe uma associação, mas um carro não precisa de passageiros para existir.
  - **Composição**: quando a parte não existe sem o todo. Como no caso do computador, que não pode existir sem um processador, disco, etc.

Universidade Federal do Amazonas
Instituto de Computação
Projeto de Programas
Técnicas Avançadas de Programação



Encapsulamento em Java

ColabWeb: bit.ly/pp-colabweb

Horácio Fernandes
horacio@icomp.ufam.edu.br

Encapsulamento na Prática

- Em Java, o encapsulamento é implementado através de uma classe (módulo) e dos **modificadores de acesso** (controle de acesso)
- Modificadores de acesso
  - Permitem controlar o acesso aos
    - *Atributos*
    - *Métodos*
    - *Classes*
    - *Interfaces*

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

Modificadores de Acesso

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

Modificadores de Acesso – Exemplo

```
package 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

Modificador `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 a partir de qualquer classe
- **Métodos** com acesso `public`:
  - São visíveis por qualquer outra classe em Java
  - Podem ser executados a partir de qualquer classe

Modificador `Public`

- **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 menos comum, é possível que um arquivo `.java` tenha uma classe pública e outras classes com outros acessos, mas esta prática não é recomendada*

Modificador `Public`

```
package 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));
  }
}
```

```
package util;
import 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) );
  }
}
```

Classe pública

Atributos públicos

Método público

Faz com que a classe `Ponto` seja observável (pacotes diferentes)

Instanciando um objeto da classe pública `Ponto`

Alterando o valor de um atributo público diretamente

Executando um método público

Modificador `Protected`

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

- **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

Modificador `Protected`

- **Classes**:
  - Se for uma classe interna (uma classe dentro de outra), a classe interna poderá ser `protected`
    - *Neste caso, terá acesso similar a um atributo da classe top-level*
  - Se for uma *top-level* class (fora da classe principal) não é possível que ela seja `protected`

Modificador `Protected`

```
package geometrico;

public class Ponto {
  protected int x, y;
}
```

```
package 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));
  }
}
```

Acessando atributos protegidos da classe atual e da superclasse

```
package util;
import 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

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

Modificador `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
  - Se forem classes internas (classe dentro de outra), podem ser `private`
  - Classes normais (top-level) não podem ser declaradas como private

Acesso `Private`: *Getters* e *Setters*

- A melhor forma de implementar encapsulamento em Java é declarando **todos os atributos** de uma classe como sendo `private`
  - Isso evita o acesso direto aos atributos da classe, **evitando que valores inválidos possam ser atribuídos** a eles
  - 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*

Acesso `Private`: *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 comuns, que o Eclipse é capaz de gerá-los automaticamente
  - Menu `Source` → `Generate Getters and Setters ...`

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

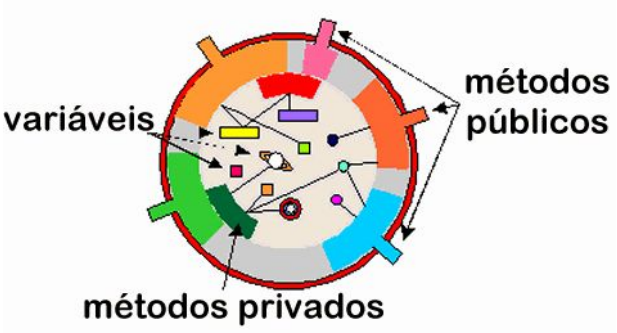
Setter para o atributo `titulo`

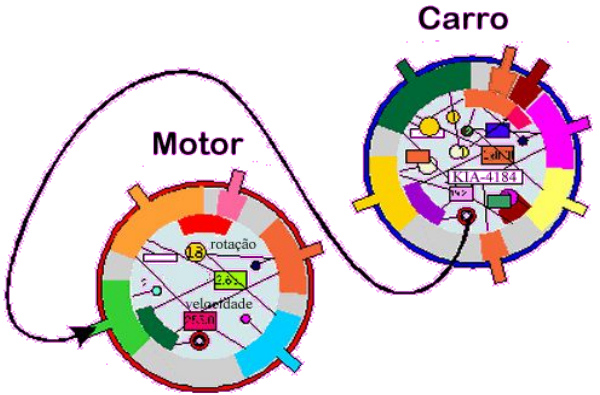
Getter para o atributo `anoPublicacao`

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

Mensagens

- 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**
    - *Esta "conversa" entre objetos é conhecida como **Troca de Mensagens***

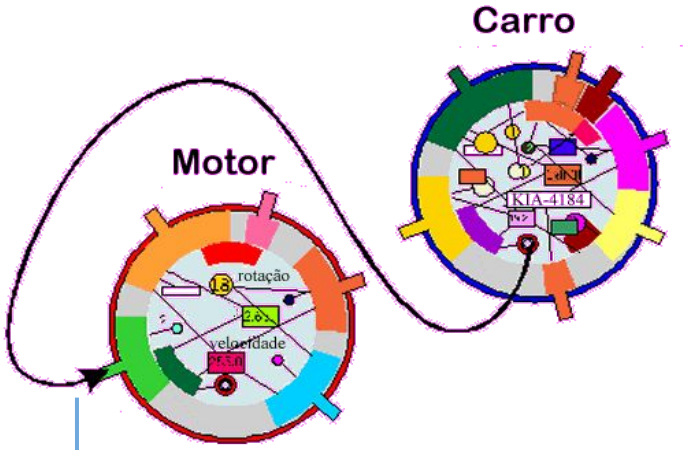




Mensagens

- Troca de Mensagens
  - É a **chamada a um método**
  - Um objeto pode chamar um método de outro objeto. Neste caso, dizemos que houve uma **troca de mensagens** entre os objetos
  - Apesar de teórico, é um conceito muito utilizado em orientação a objetos

```
class Carro {
  // ...
  public void andar() {
    motor.acelerar();
  }
  // ...
}
```



Troca de mensagem

Modificadores - Resumo

Modificador	Classe	Pacote	Subclasse (mesmo pacote)	Subclasse (dif. pacotes)	Mundo
`public`	✔	✔	✔	✔	✔
`protected`	✔	✔	✔	✔	
nenhum	✔	✔	✔		
`private`	✔				

Laboratório

- Disponível no ColabWeb
  - bit.ly/pp-colabweb