

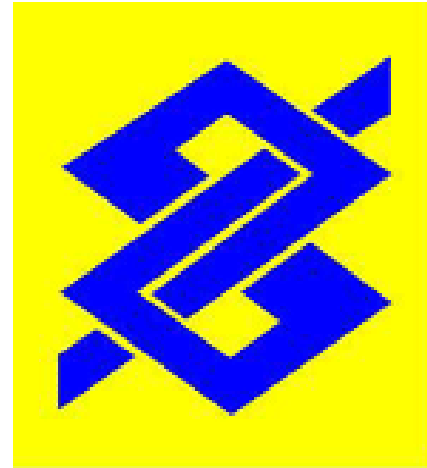
O que é Objeto?

# Objetos

- Podem ser físicos ou conceituais
  - Conceitual
  - Físico

# Objetos

- Conceituais
  - Conta Corrente



# Objetos

- Físicos
  - Celular



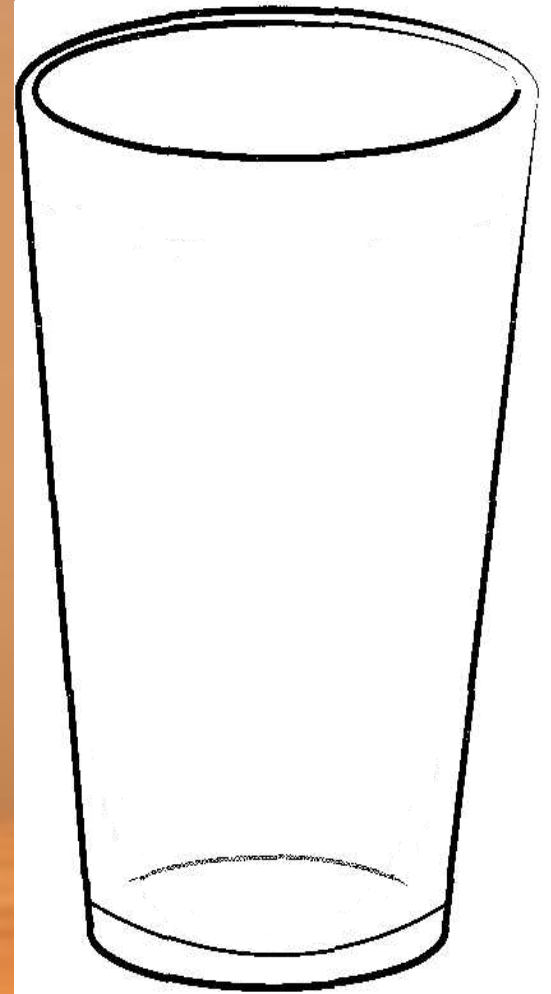
# Atributos

- Características



|         |          |          |
|---------|----------|----------|
| AMARELO | AZUL     | LARANJA  |
| PRETO   | VERMELHO | VERDE    |
| ROXO    | AMARELO  | VERMELHO |
| LARANJA | VERDE    | PRETO    |
| AZUL    | VERMELHO | ROXO     |
| VERDE   | AZUL     | LARANJA  |

# Estados



# Operações

- Conhecidas como Comportamento
- Coisas que os objetos podem fazer
- Normalmente afetam os atributos de um objeto

# Programação Orientada a Objetos em C#



# Programação Orientada a Objetos (POO): Definição

“Programação Orientada a Objetos é um método de implementação no qual programas são organizados como uma coleção de **objetos** cooperativos, onde cada um deles representa um **instância** de alguma classe, e cujas classes são membros de uma hierarquia de classes unificada por suas **relações de herança**.”  
(Booch, 1994).

**O que é uma Classe?**

**Qual a relação entre Classe e Objeto?**

# Programação Orientada a Objetos: Recursos

As linguagens orientadas a objeto são caracterizadas pelo suporte a quatro recursos chaves (Ghezzi, 1997):

- definição de tipo de dados abstratos;
- herança;
- polimorfismo;
- vinculação dinâmica.

**Qual a relação entre “tipo de dados abstratos” e classes?**

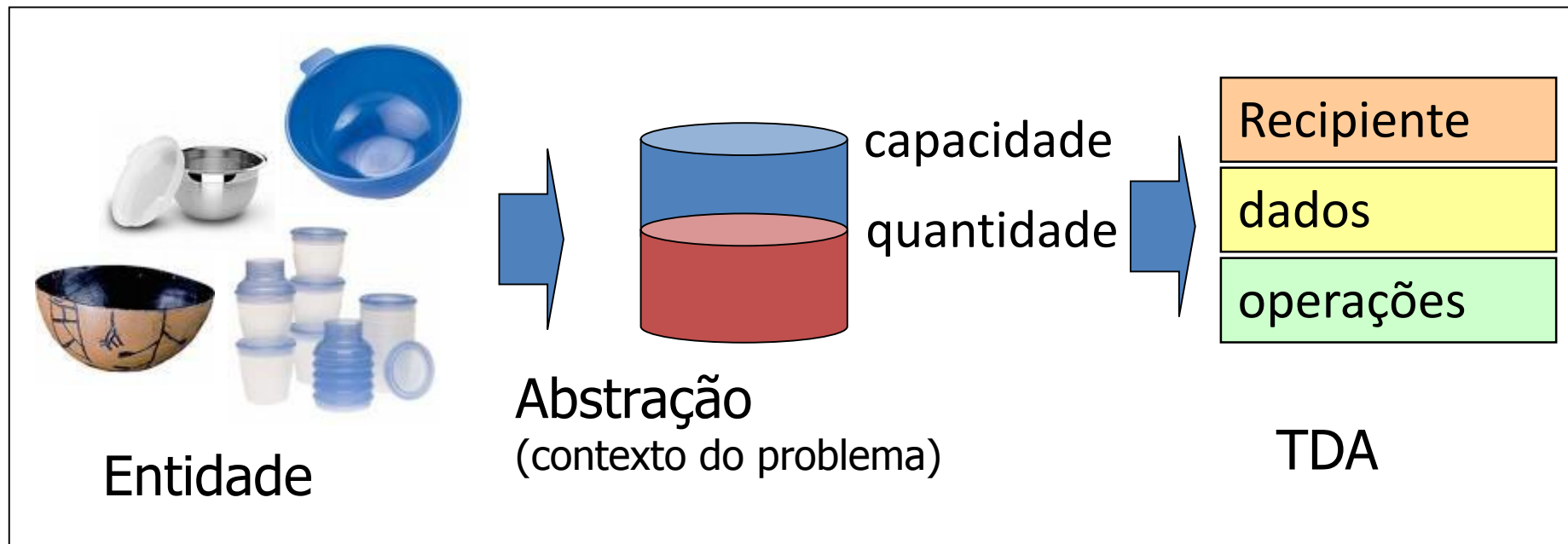
# Tipos de Dados Abstratos

“Uma **abstração** é uma visualização ou uma representação de uma entidade que inclui somente os atributos de importância em um contexto particular.” (Sebesta, 2000)

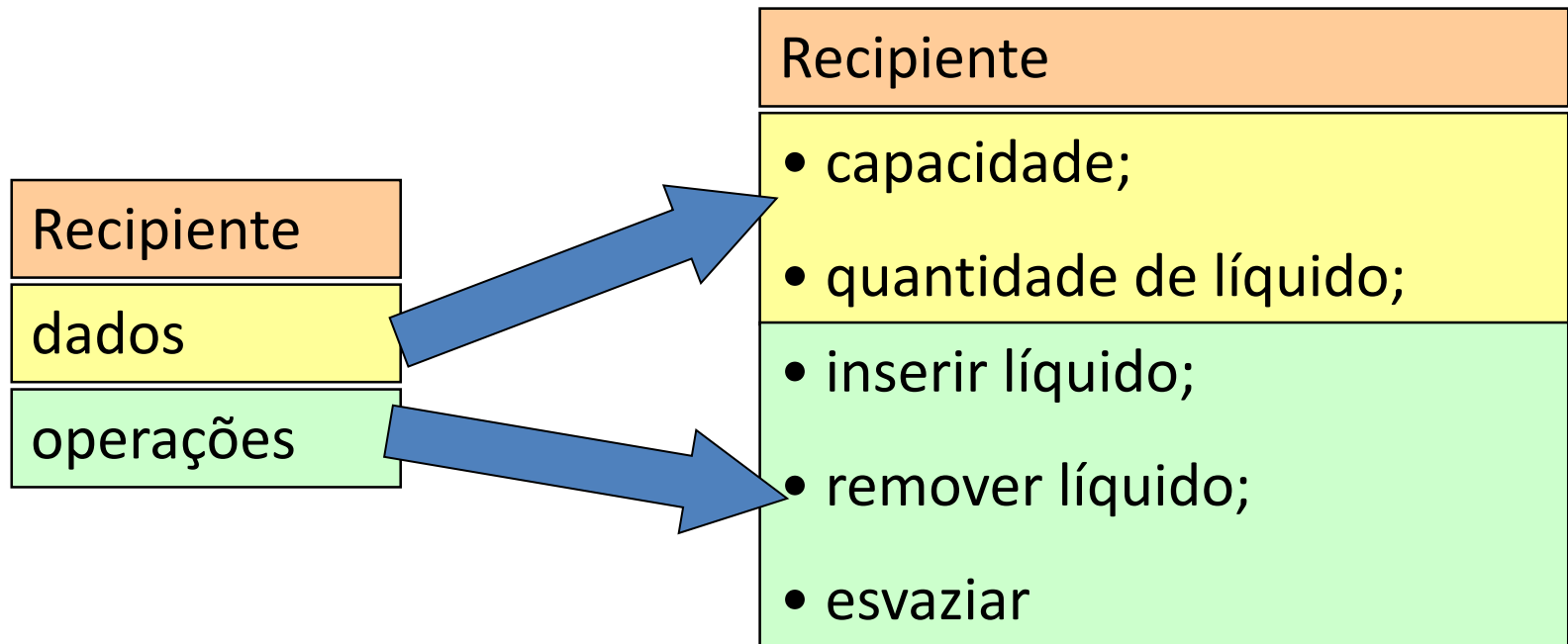
“Um **tipo de dado** significa um conjunto de valores e uma seqüência de operações sobre estes valores. Este conjunto e estas operações formam uma construção matemática que pode ser implementada usando uma determinada estrutura de dados do hardware ou do software.” (Tenenbaum, 1995)

# Tipos de Dados Abstratos

O TDA pode ser visto como uma forma de especificação das características relevantes das entidades envolvidas no problema, de que forma elas se relacionam e como podem ser manipuladas.



# Tipos de Dados Abstratos



# Tipos de Dados Abstratos

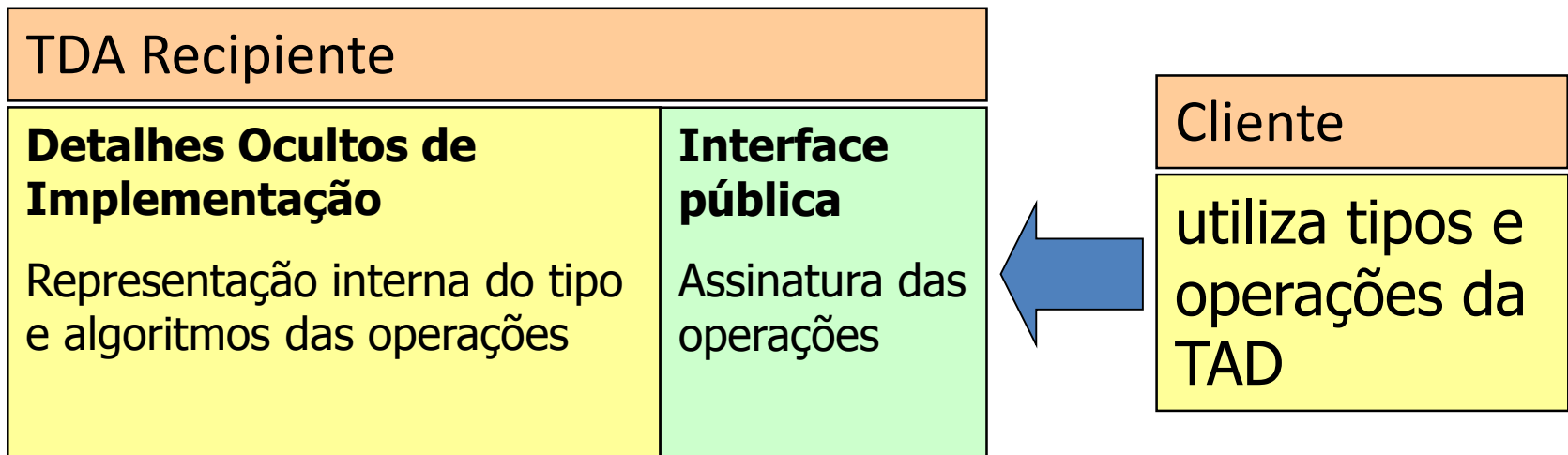
Em geral, o tipo de dado abstrato é implementado satisfazendo duas condições:

- A representação ou a definição do TDA e as operações sobre os objetos do tipo estão contidas em uma única unidade sintática. Além disso, outras unidades de programação podem ter permissão para criar variáveis do tipo definido.
- A representação de objetos do TDA não é visível pelas unidades de programa que usam o tipo (clientes), de modo que as únicas operações diretas possíveis sobre esses objetos são aquelas oferecidas na definição do tipo (**ocultação de informação**).

Essas condições favorecem o **Encapsulamento** do tipo de dado.

# Tipo de Dados Abstratos

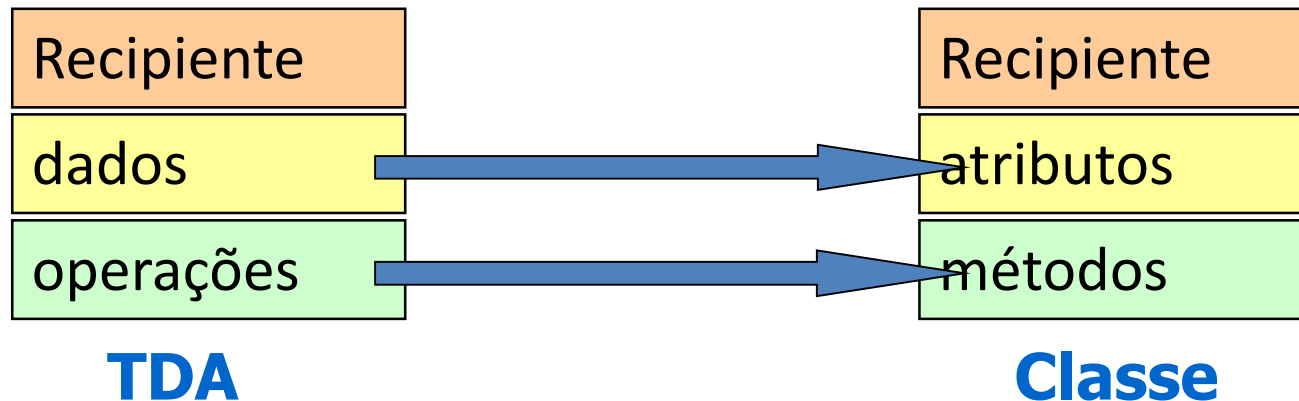
Seguindo estas condições, as unidades clientes não são dependentes do tipo de implementação do TDA, o que possibilita que a implementação do TDA seja alterada sem comprometer as unidades clientes.



# Conceito de Classe

Nas linguagens de programação orientadas a objetos temos a evolução do conceito de tipo de dados abstratos (dados + operações) ampliando a capacidade de reutilização do TDA em outros problemas.

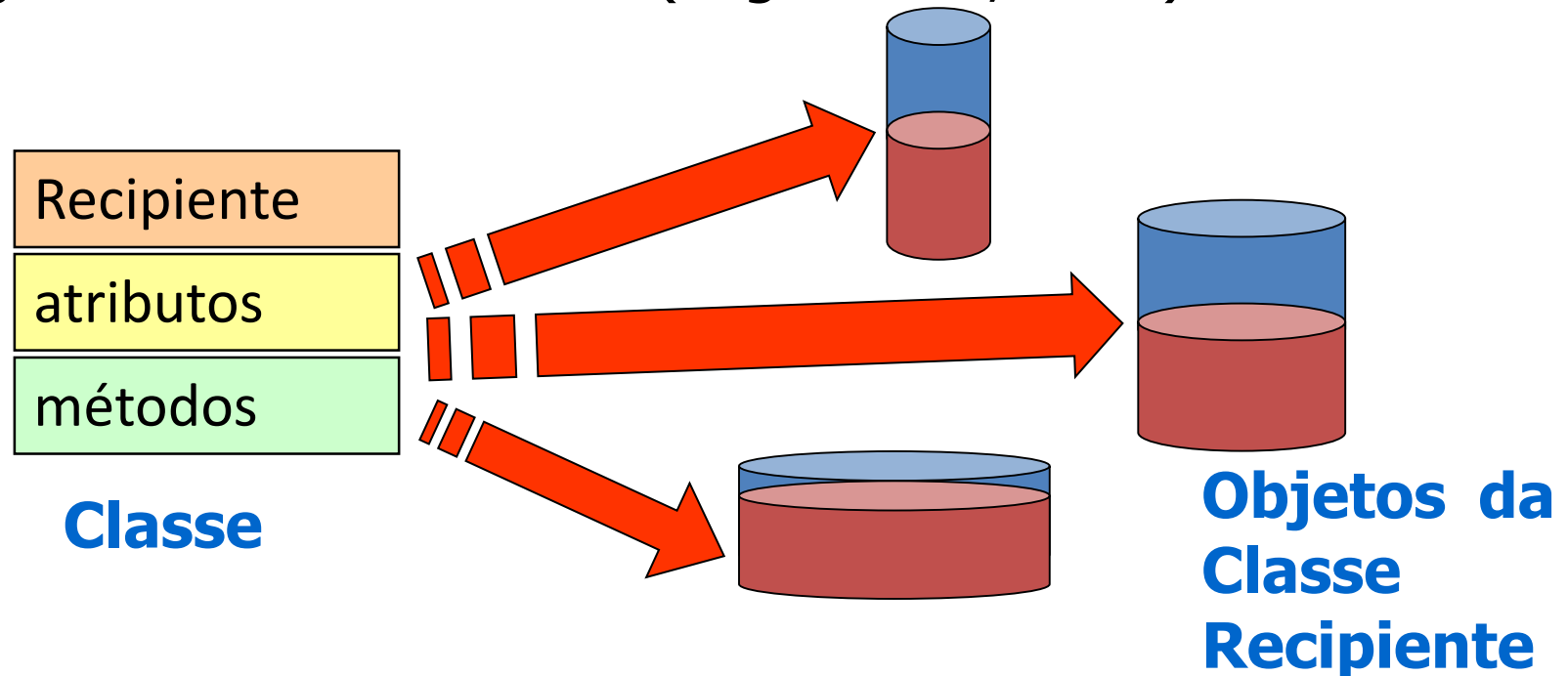
No contexto de implementação, o conceito de TDA ampliado em Linguagens OO está relacionado diretamente ao conceito de Classe.



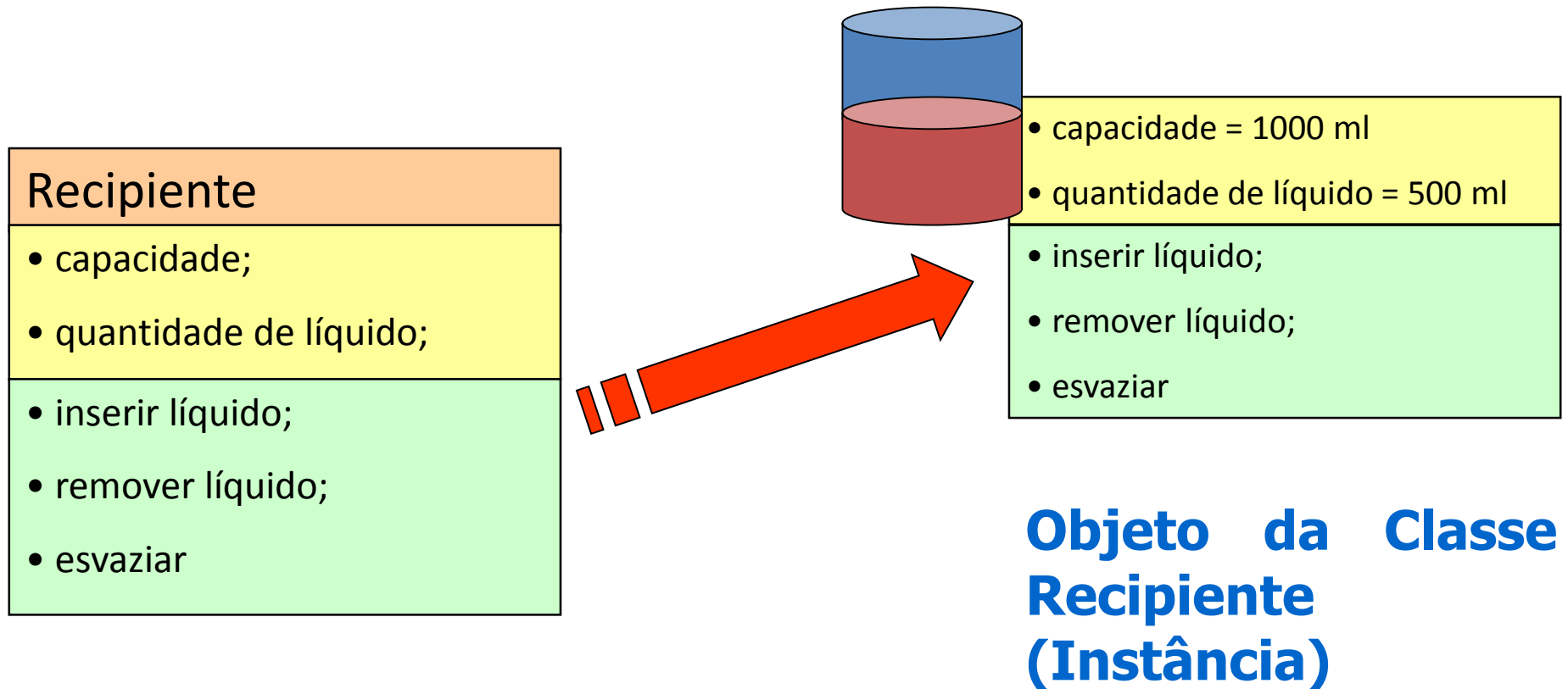


# Relação entre Classe e Objeto

“Uma classe é o estêncil (forma) a partir do qual são criados (gerados) objetos. Cada objeto tem a mesma estrutura e comportamento da classe na qual ele teve origem. Se o objeto obj pertence à classe C, dizemos que “obj é uma instância de C” (Page-Jones, 2001)



# Relação entre Classe e Objeto

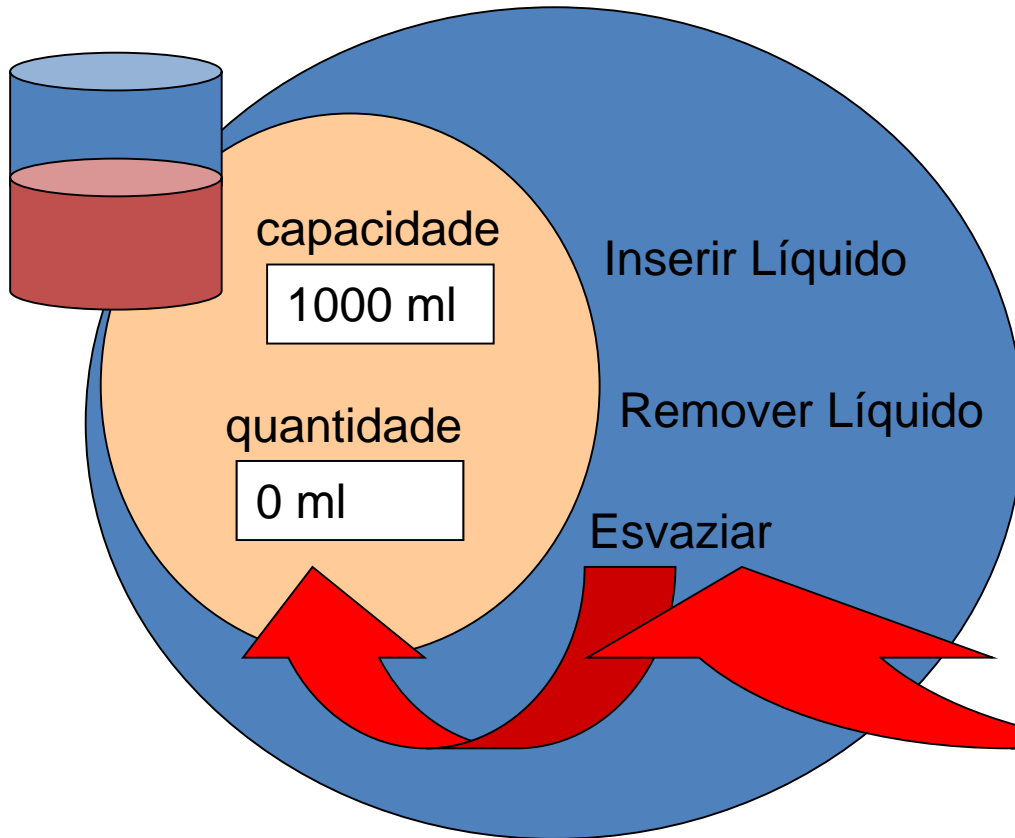


**Classe**

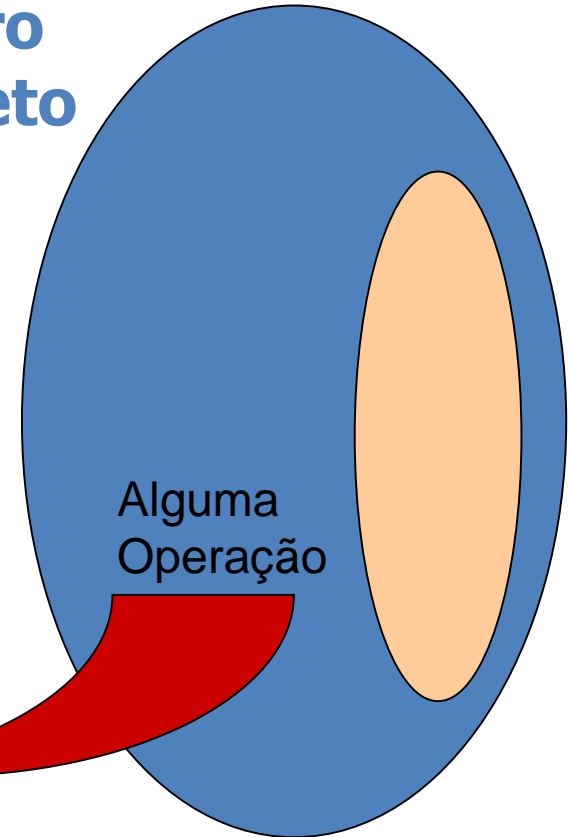
**Objeto da Classe  
Recipiente  
(Instância)**

# Comportamento do Objeto

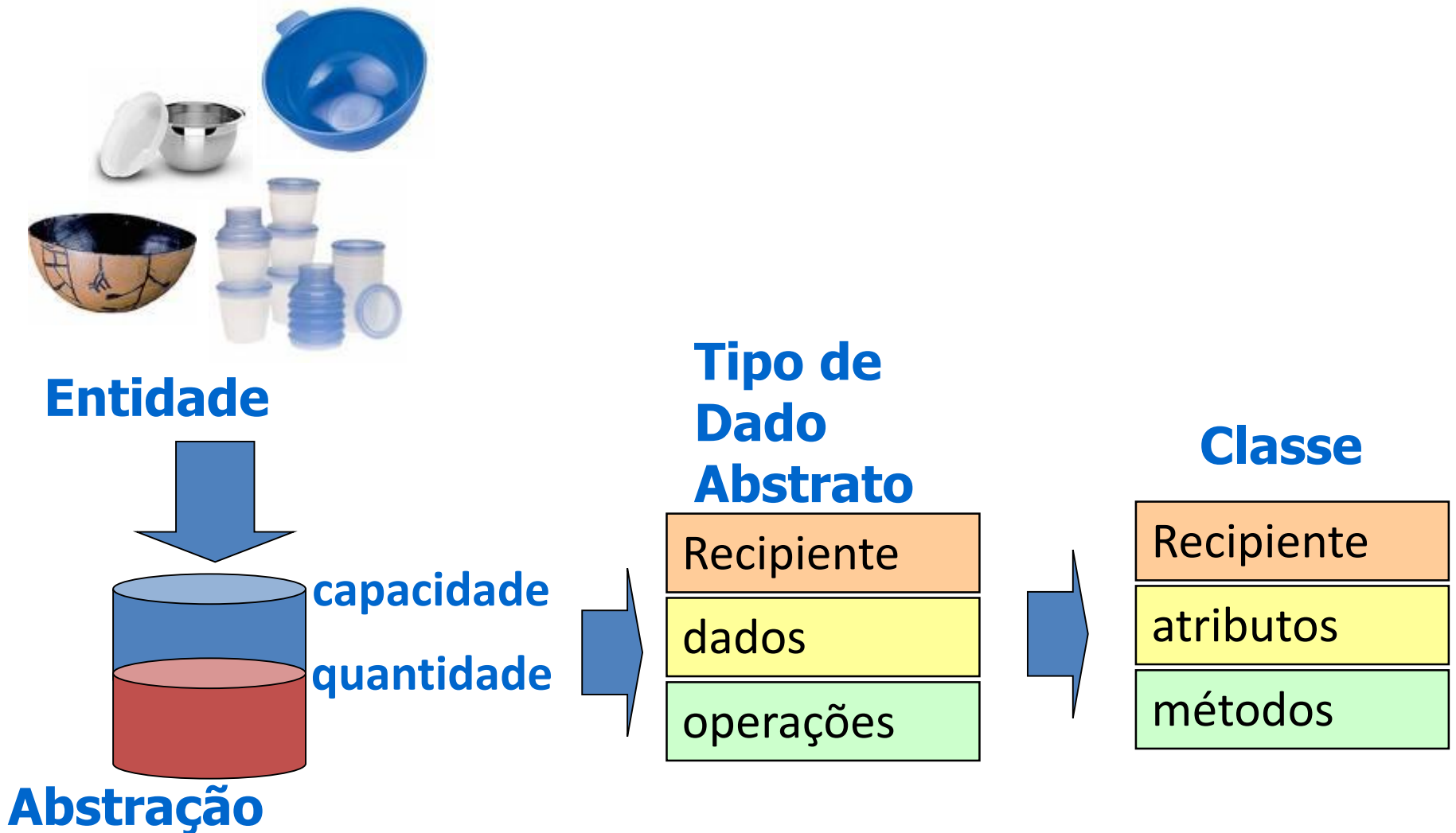
**Objeto**



**Outro  
Objeto**



# Implementação de um TDA



# Sintaxe de uma classe em C#

## Classe

|            |
|------------|
| Recipiente |
| atributos  |
| métodos    |

```
<modificador de acesso> class <nome> {
```

**<Atributos:** armazenam informações. >

**<Métodos Construtores:** métodos especiais que são chamados no momento da criação de um objeto da classe. >

**<Métodos:** métodos que descrevem operações sobre a classe e seus objetos. >

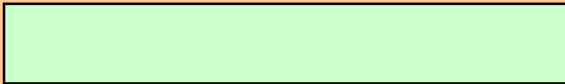
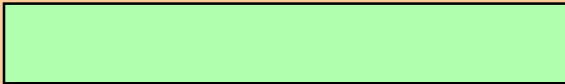
```
}
```

A ordem entre os componentes internos da classe não precisa ser a mesma apresentada na figura.

# Exemplo

arquivo Recipiente.cs

```
public class Recipiente {
```



```
}
```

Por “padrão”, o nome da classe tem apenas a primeira letra maiúscula. O C#, assim como o C, faz distinção entre letras maiúsculas e minúsculas.

Padrão: Upper CamelCase

O modificador de acesso *public* define que a classe pode ser utilizada por qualquer outra classe.

# Atributos

```
<modificador de acesso> <tipo> <nome>;
```

```
public class Recipiente {  
    private int capacidade, quantidade;  
      
      
      
}
```

Por “padrão”, os nomes dos atributos são escritos em letra minúscula.

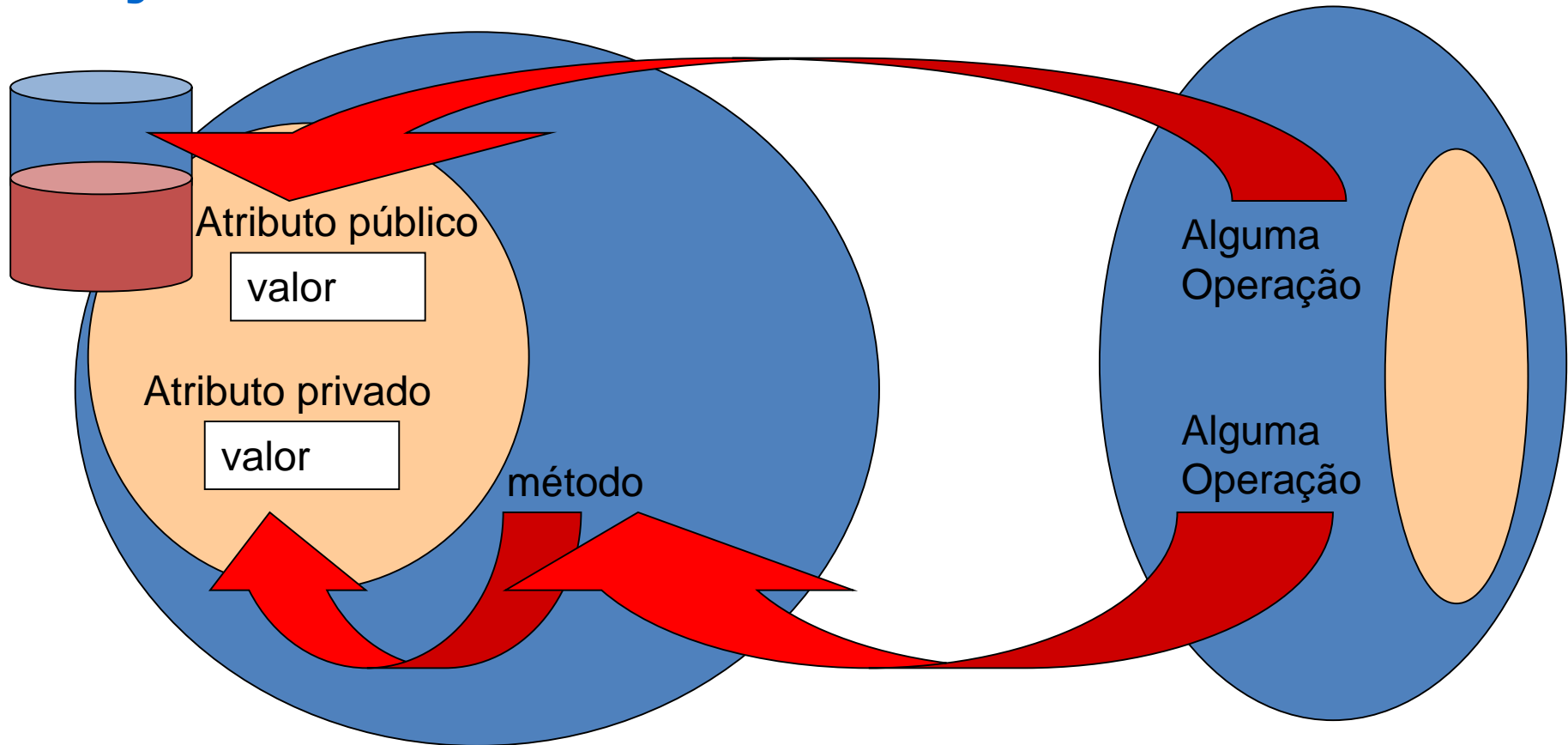
Padrão: Lower CamelCase

O modificador ***private*** define que a visibilidade do atributo será limitada apenas para o uso dentro da classe. Já o ***public*** libera a visibilidade também para as outras classes. A ausência do modificador de acesso indica acesso de **pacote**, ou seja, acesso apenas pelas classes dentro do mesmo pacote.

# Modificador de acesso

**Objeto**

**Outro  
Objeto**





# Exemplo de classe com atributos

```
/* Esta classe foi desenvolvida para modelar um recipiente  
que pode armazenar líquidos de acordo com sua capacidade  
*/
```

```
public class Recipiente {
```

```
    private int capacidade; //capacidade do recipiente  
    private int quantidade; //quantidade de líquido atual  
    private String marca; //marca do recipiente
```

```
}
```

# Métodos

```
<modificador de acesso> <tipo de retorno> <nome> ( <lista de parâmetros> ) {  
}
```

```
public class Recipiente {
```

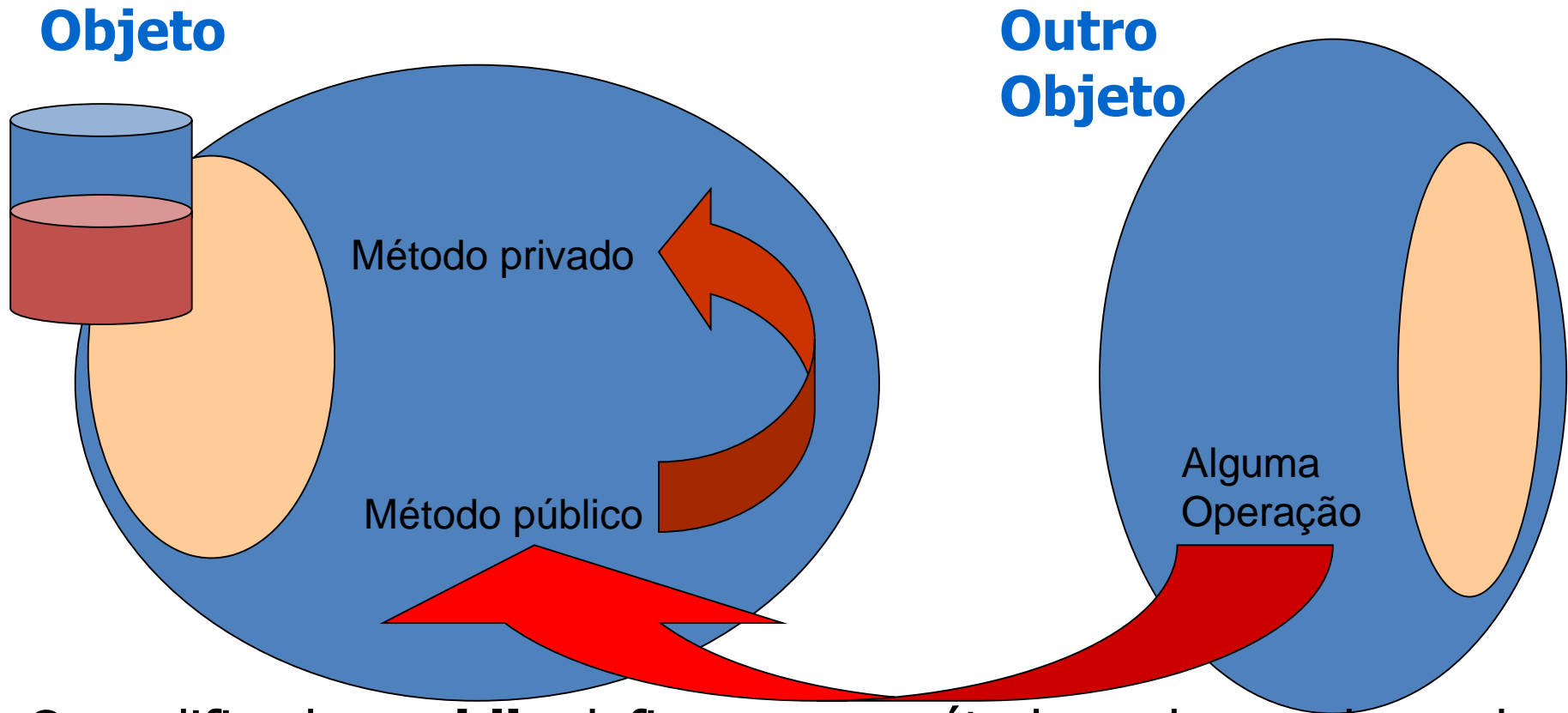
```
    public int ObterQuantidade() {  
        return quantidade; // retorna o valor do atributo  
    }
```

```
    public void AtribuirQuantidade (int qtde) {  
        quantidade=qtde; // atribui o valor do parâmetro à quantidade  
    }  
}
```

Por “padrão”, os nomes dos métodos tem apenas a primeira letra maiúscula. Nomes compostos tem a letra maiúscula para evitar espaços.

**Padrão: Upper CamelCase**

# Modificador de acesso



O modificador **public** define que o método pode ser chamado por qualquer classe. Já o **private** torna a chamada restrita apenas dentro da classe. A ausência do modificador define acesso de interno.

# Retorno dos Métodos

```
public int ObterQuantidade () {  
    return quantidade; // retorna o valor do atributo quantidade  
}
```

```
public void AtribuirQuantidade (int qtde) {  
    quantidade=qtde, // atribui o valor do parâmetro à capacidade  
}
```

Quando um tipo de retorno é definido, retornar um valor é obrigatório (*return*). A palavra *void* indica que o método não tem retorno.

\*O comando *return* deve ser o último comando executado dentro de um fluxo de execução, pois quando é encontrado sai do método.

# Parâmetros

```
public void AtribuirQuantidade ( int qtde ) {  
    quantidade=qtde, // atribui o valor do parâmetro à quantidade  
}
```

Um método pode ter mais de um parâmetro e são especificados em seqüência separados por vírgula.

```
public void atribuirCapacidadeQuantidade ( int cap, int qtde ) {  
    capacidade=cap; // atribui o valor do parâmetro à capacidade  
    quantidade=qtde, // atribui o valor do parâmetro à quantidade  
}
```

# Exemplo

arquivo Recipiente.cs

```
/* Esta classe foi desenvolvida para modelar um recipiente que pode armazenar líquidos de acordo com sua capacidade */
```

```
public class Recipiente {
```

```
    private int capacidade; //capacidade do recipiente  
    private int quantidade; //quantidade de líquido
```

```
    atual
```

```
    public int ObterQuantidade () {  
        return quantidade; // retorna o valor do atributo quantidade  
    }
```

```
    public void AtribuirQuantidade (int qtde) {  
        quantidade=qtde; // atribui o valor do parâmetro à quantidade  
    }
```

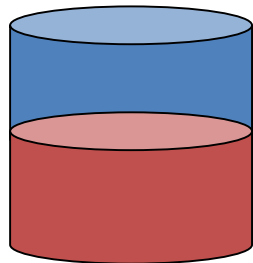
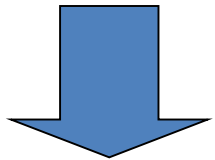
```
    outros métodos...
```

```
}
```

# Criação de Objetos



**Entidade**



capacidade  
quantidade



**Tipo de  
Dado  
Abstrato**

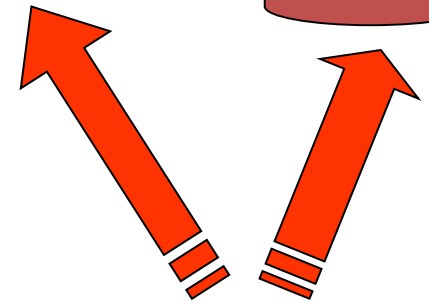
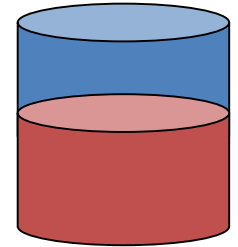
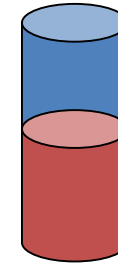
|            |
|------------|
| Recipiente |
| dados      |
| operações  |



**Classe**

|            |
|------------|
| Recipiente |
| atributos  |
| métodos    |

**Objetos**



# Criação de Objetos

```
public class Recipiente {
```

```
    private int capacidade;  
    private int quantidade;
```

```
    public Recipiente (int cap){  
        capacidade=cap;  
        quantidade=0; //não é necessário  
    }
```

```
    public boolean Adicionar (int qtde) {  
        if (capacidade-quantidade>= qtde) {  
            quantidade+=qtde;  
            return true;  
        }else  
            return false;  
    }
```

```
    public boolean Remover (int qtde) {...}
```

```
    public void Esvaziar ( ) {...}
```

```
}
```

```
public class Cliente {
```

```
    public void Método ( ) {
```

```
        Recipiente objA = new Recipiente (350);  
        boolean sucess;
```

```
        sucess = objA.a...nar (50);  
        sucess = objA...er (30);
```

```
    }
```

O comando **new** é responsável em solicitar a criação de um objeto na memória dinamicamente.



# Método Construtor

```
public class Recipiente {
```

```
    private int capacidade;  
    private int quantidade;
```

```
    public Recipiente (int cap){  
        capacidade=cap;  
        quantidade=0;//não é necessário  
    }
```

```
    public boolean Adicionar (int qtde) {  
        if (capacidade-quantidade>= qtde) {  
            quantidade+=qtde;  
            return true;  
        } else  
            return false;  
    }
```

```
    public boolean Remover (int qtde) {...}
```

```
    public void Esvaziar ( ) {...}  
}
```

O método construtor possui sempre o mesmo nome que a classe e é chamado quando um objeto é criado. Este método nunca possui retorno. Assim, não é colocada nem a palavra *void*.

```
Recipiente objA = new Recipiente (350);
```

# Método Construtor

```
public class Recipiente {
```

```
    private int capacidade;  
    private int quantidade;
```

```
    public Recipiente (int cap){  
        capacidade=cap;  
    }
```

```
    public boolean Adicionar (int qtde) {  
        if (capacidade-quantidade>= qtde) {  
            quantidade+=qtde;  
            return true;  
        }else  
            return false;  
    }
```

```
    public boolean Remover (int qtde) {...}
```

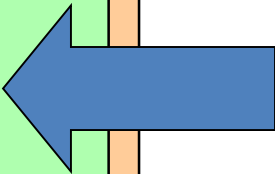
```
    public void Esvaziar ( ) {...}  
}
```

Os atributos numéricos da classe são inicializados com 0 por default, os booleanos com false, e as referências com null (referência nula).

# Método Construtor

```
public class Recipiente {
```

```
    private int capacidade;  
    private int quantidade;
```

```
    public Recipiente ( ){  
        capacidade=1000;  
    }  
    
```

```
    public boolean Adicionar (int qtde) {  
        if (capacidade-quantidade >= qtde) {  
            quantidade+=qtde;  
            return true;  
        }else  
            return false;  
    }
```

```
    public boolean Remover (int qtde) {...}
```

```
    }  
    public void Esvaziar ( ) {...}
```

O construtor pode não ter parâmetros.

Se nenhum construtor for definido na classe, é gerado internamente um construtor padrão sem parâmetros.

# Método Construtor

```
public class Recipiente {
```

```
    private int capacidade = 1000;  
    private int quantidade;
```

```
    public boolean Adicionar (int qtde) {  
        if (capacidade-quantidade >= qtde) {  
            quantidade += qtde;  
            return true;  
        } else  
            return false;  
    }
```

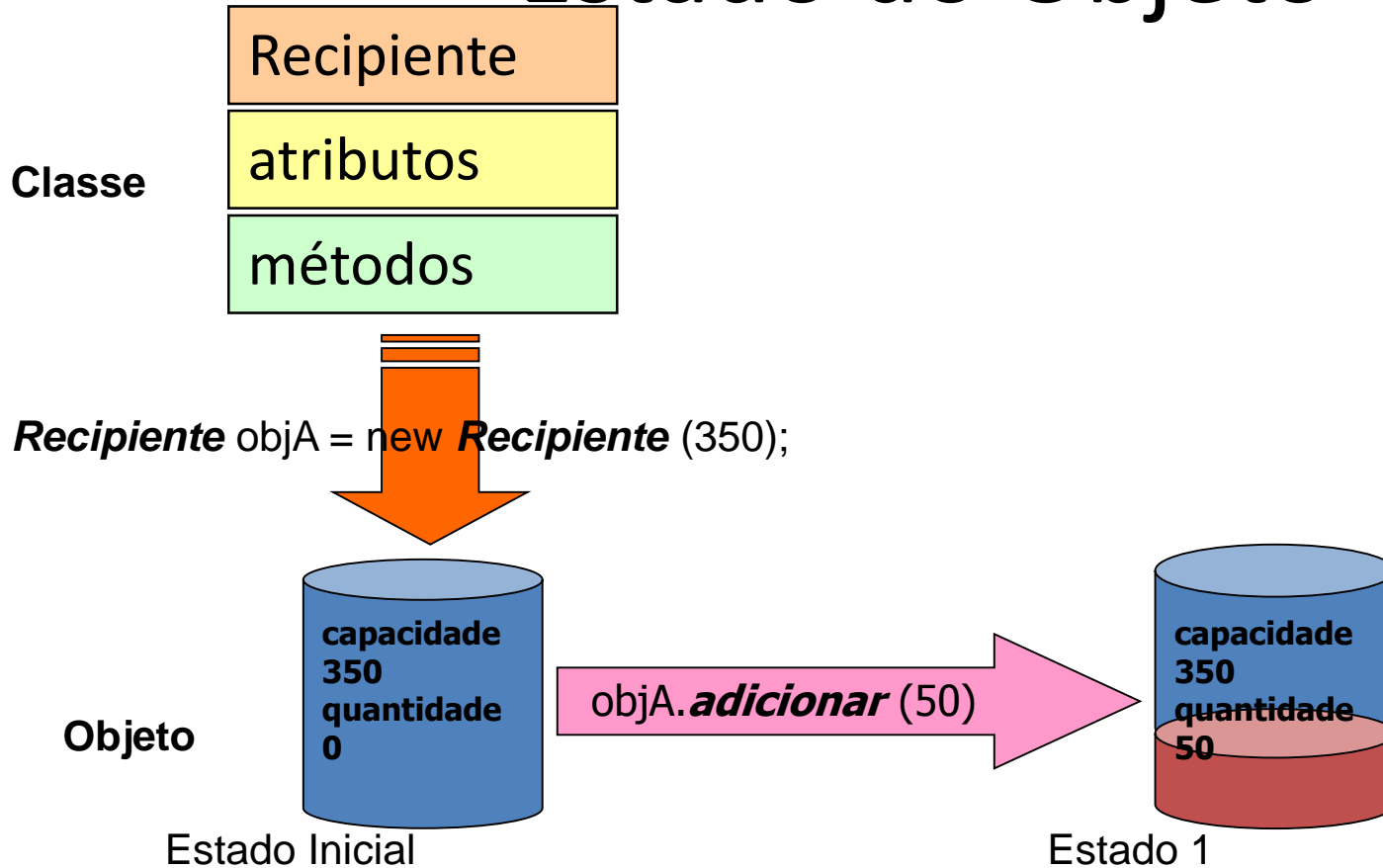
```
    public boolean Remover (int qtde) {...}
```

```
    public void Esvaziar ( ) {...}
```

```
}
```

Valores iniciais dos atributos podem ser atribuídos diretamente junto à declaração.

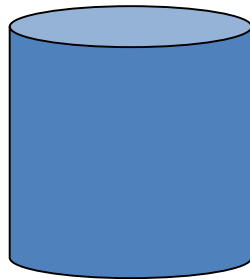
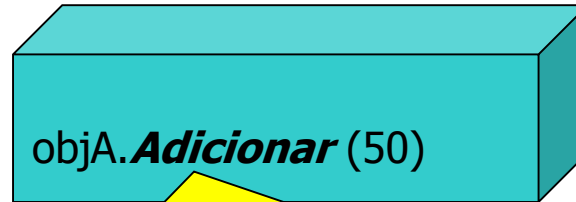
# Estado do Objeto



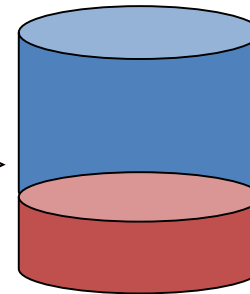
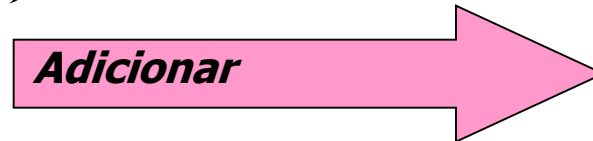
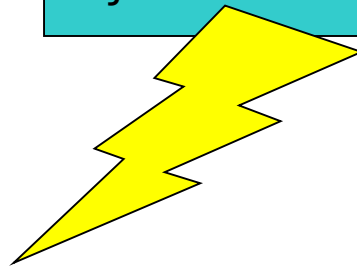
**O conjunto de valores dos atributos de um objeto em um momento específico determina seu estado. Assim, ao longo de sua existência, o objeto vai mudando de estado de acordo com os métodos aplicados.**

# Mensagens

**Objeto Cliente**



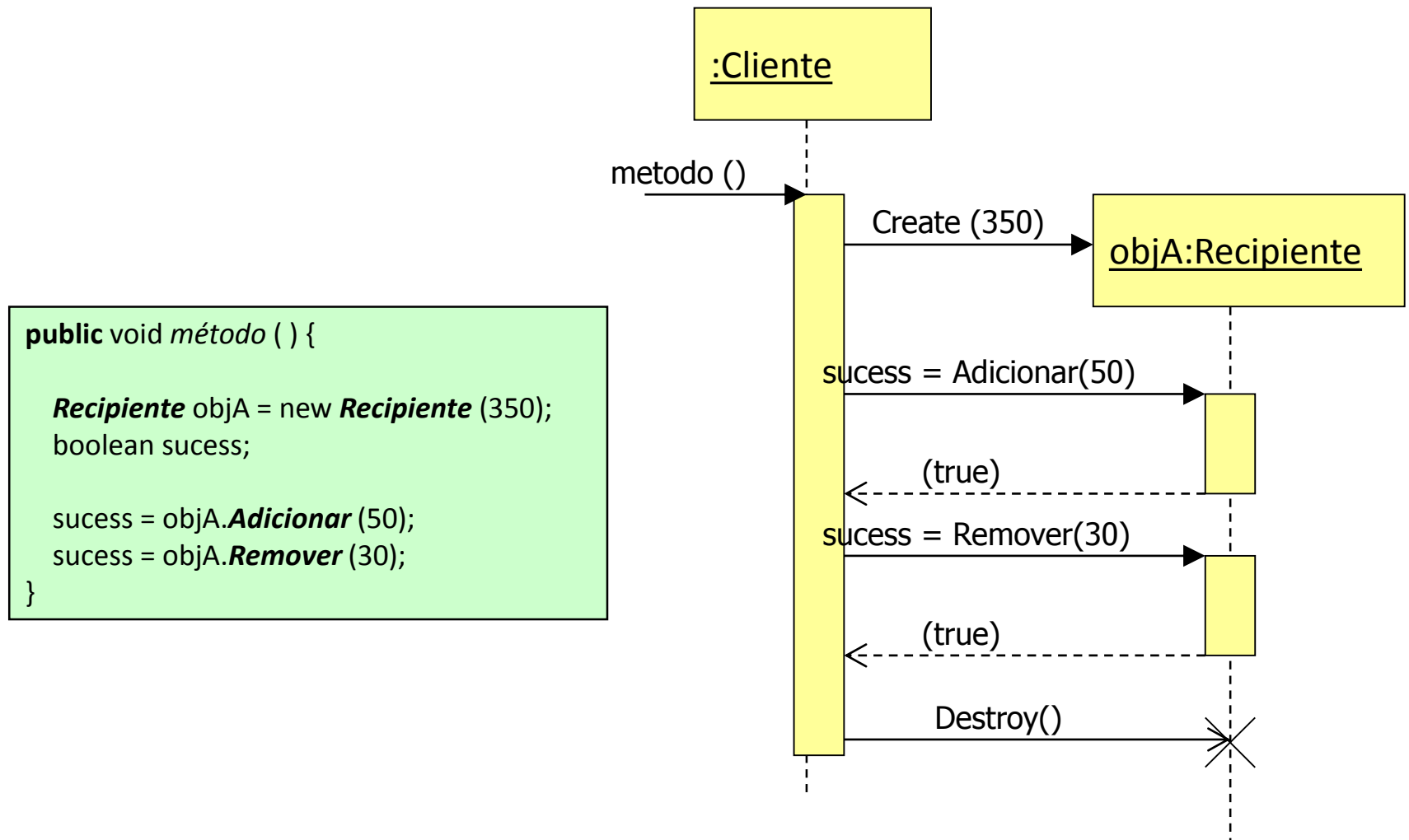
**Estado Inicial**



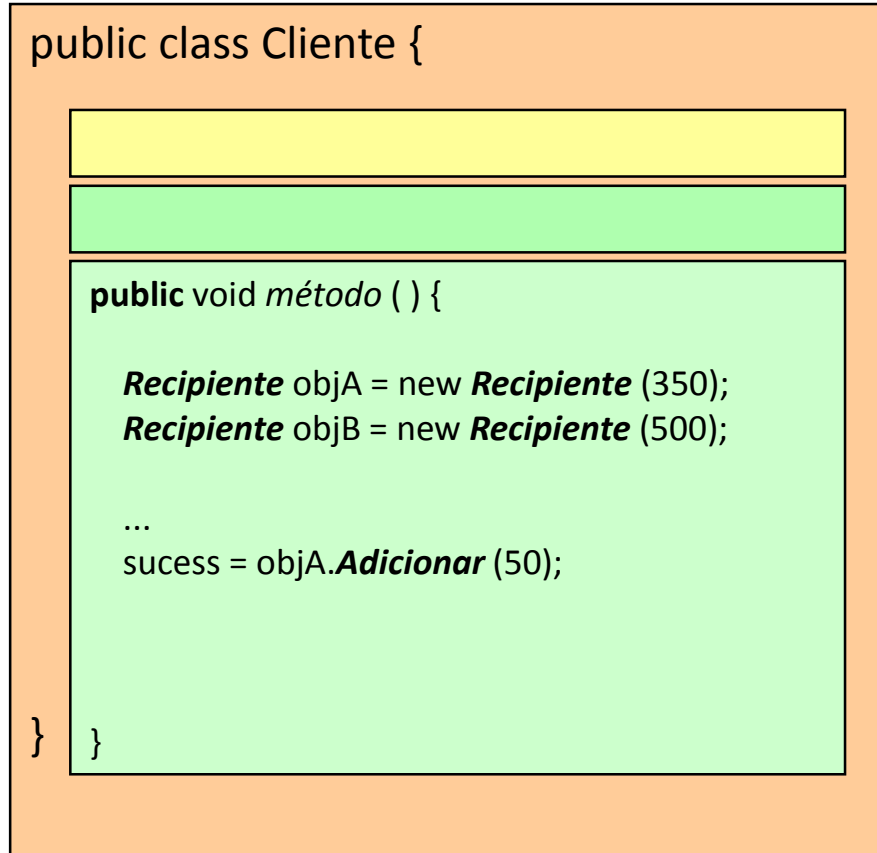
**Estado 1**

**“Uma mensagem é o veículo pelo qual um objeto remetente obj1 transmite a um objeto destinatário obj2 um pedido para o obj2 aplicar um de seus métodos.” (Page-Jones, 2001)**

# Exemplo de Diagrama de Seqüência (UML)



# Variáveis de Referência

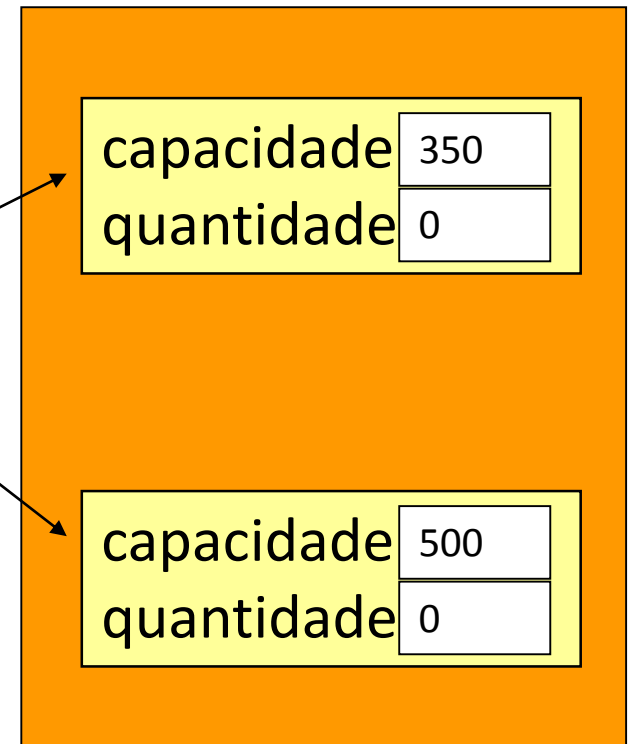
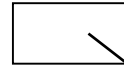


Variáveis  
de  
Referência

objA



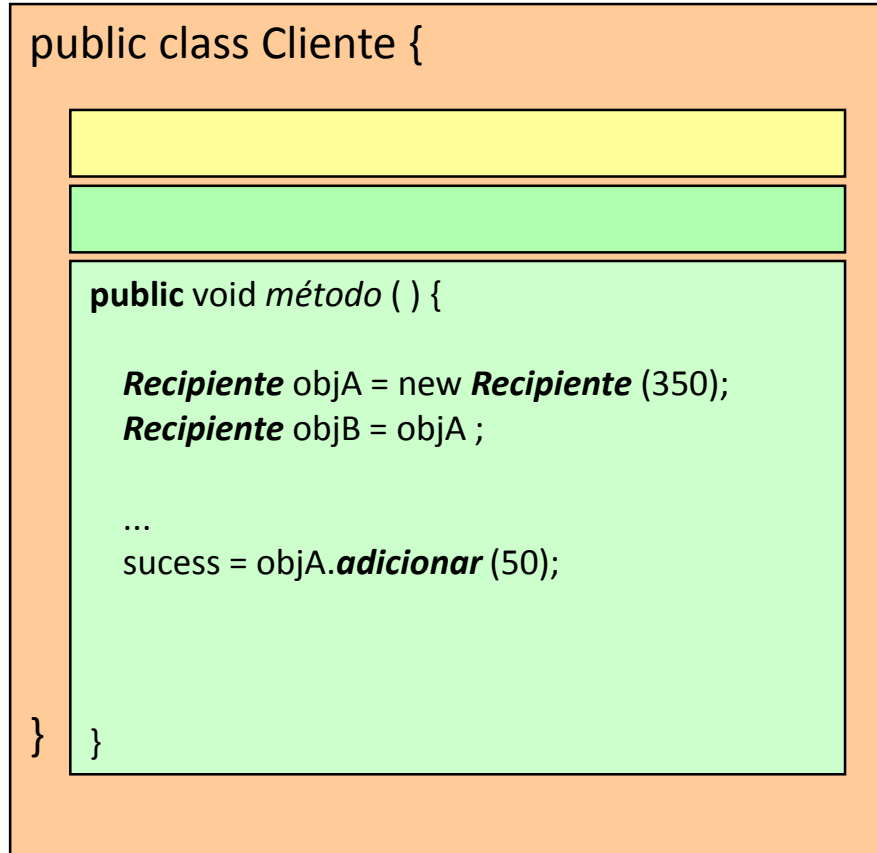
objB



Espaço de memória  
administrado pela .Net,  
denominado HEAP



# Variáveis de Referência

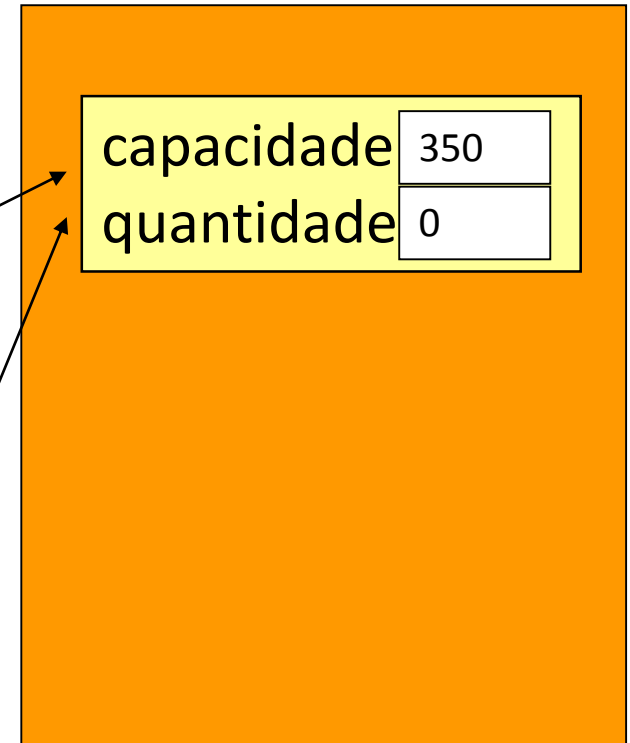
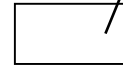


Variáveis  
de  
Referência

objA

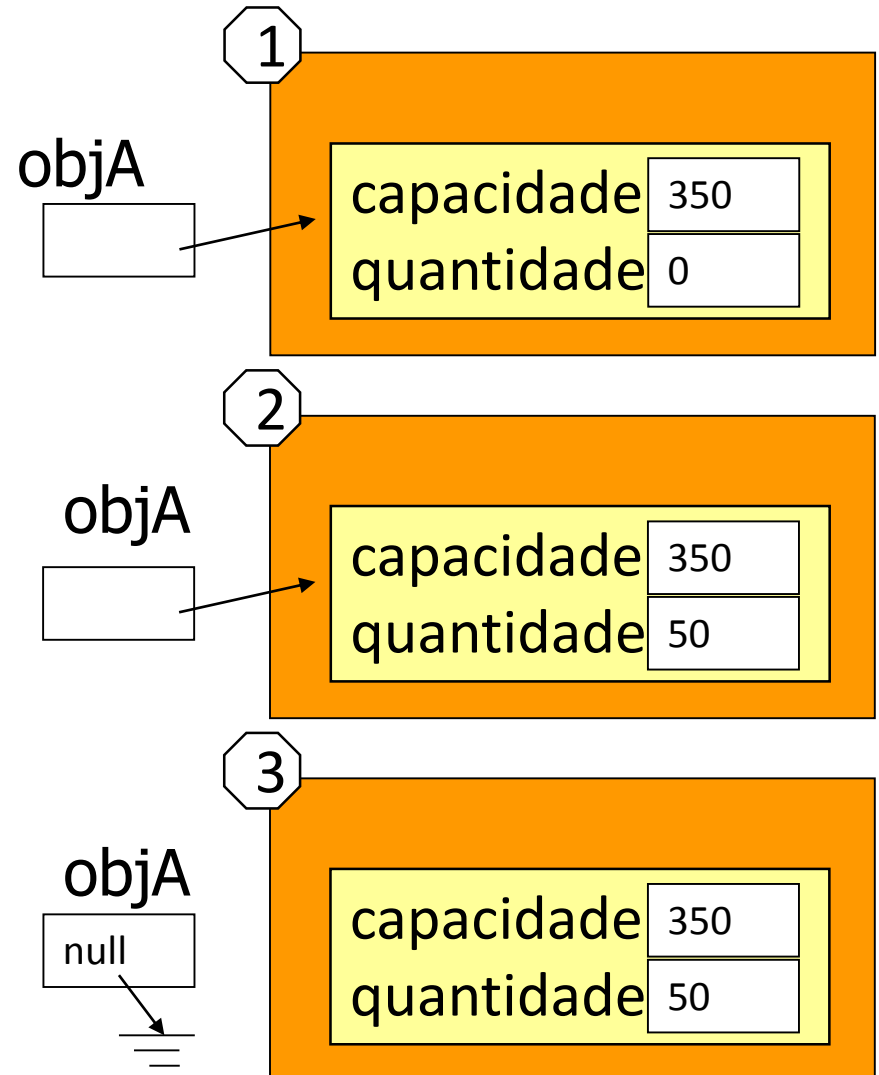
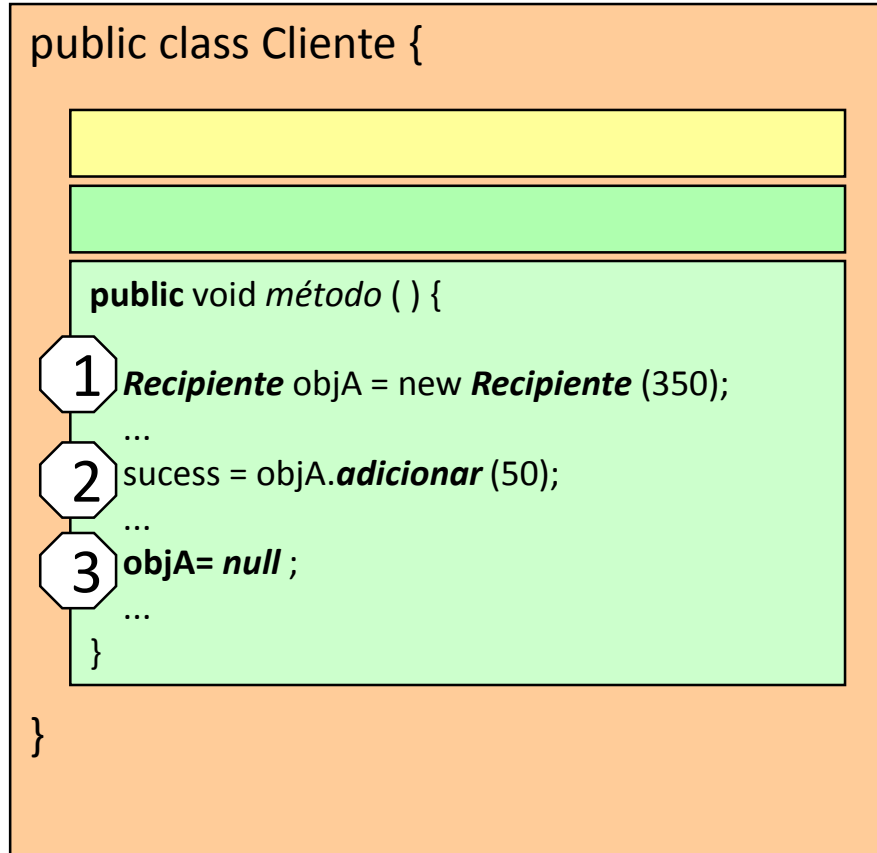


objB

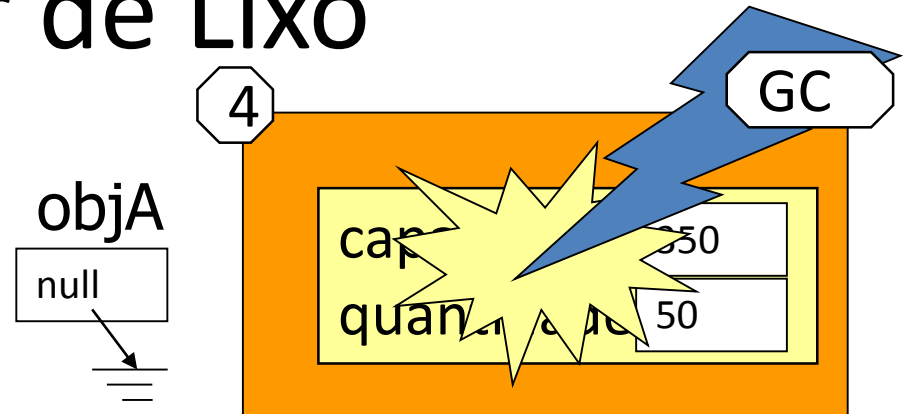
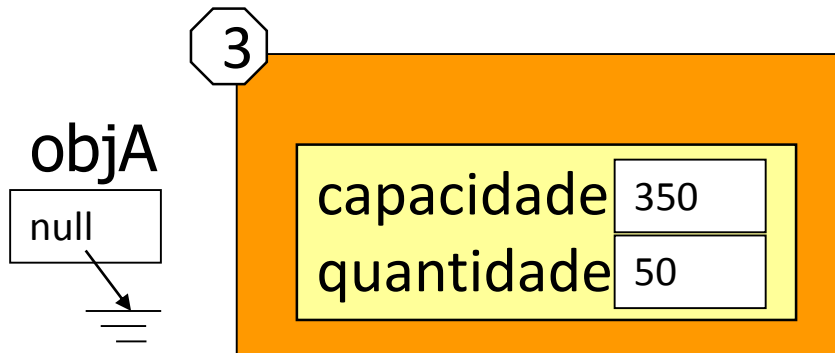
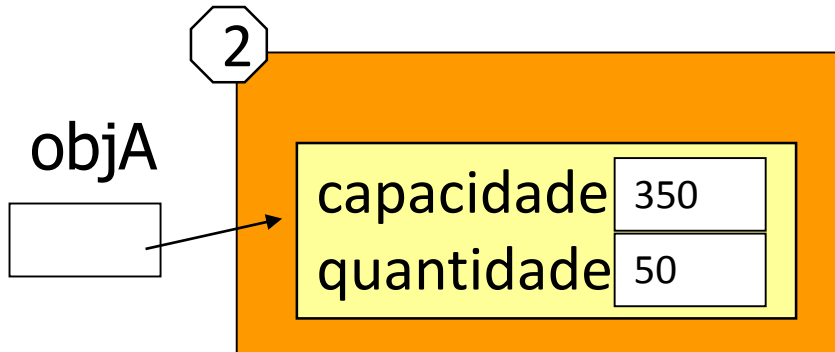
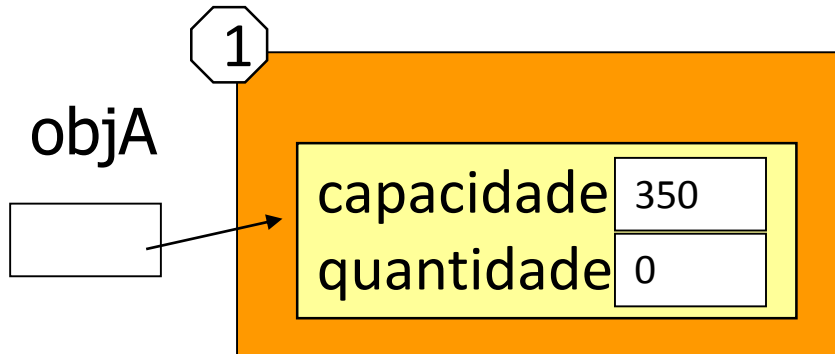


Espaço de memória  
administrado pela JVM,  
denominado HEAP

# Mudança de Estado na memória



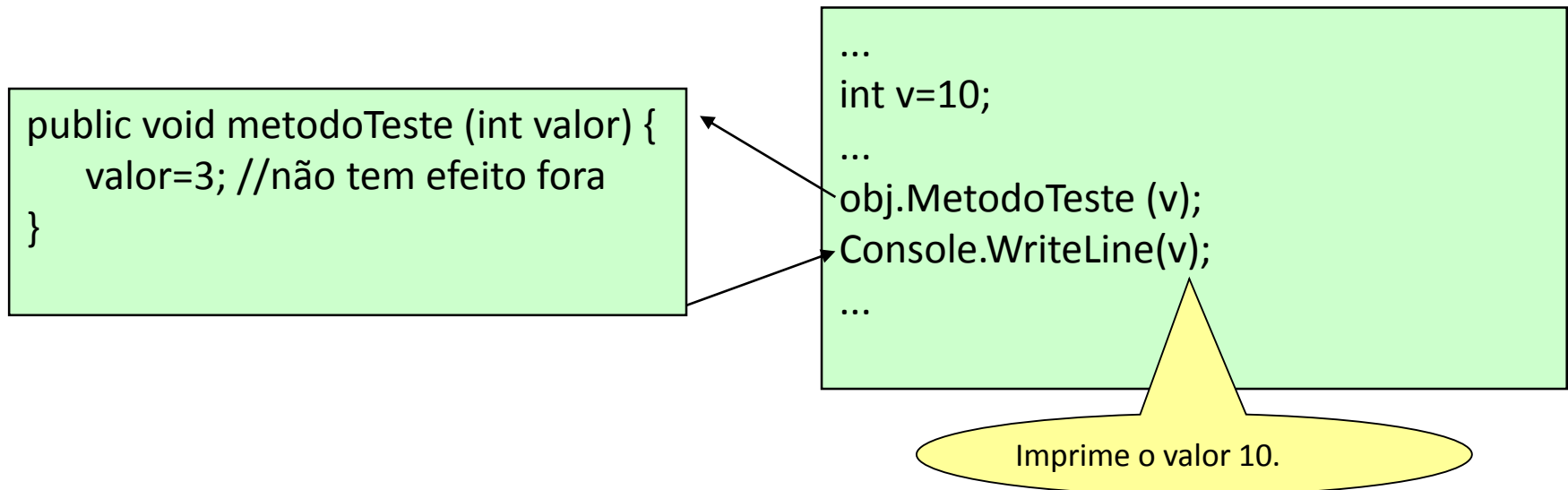
# Coletor de Lixo



**O coletor de lixo (garbage collector) é um processo executado pelo .Net com o objetivo de liberar o espaço alocado para um objeto quando este torna-se inalcançável.**

# Parâmetros de Tipos Primitivos

O mecanismo de passagem de parâmetros de tipos primitivos na linguagem C# é por cópia. Ou seja, o valor do argumento é copiado para o parâmetro, assim qualquer alteração do parâmetro não afeta o argumento.



# Parâmetros de Tipos Construídos

No caso de parâmetros de tipos construídos é realizada uma cópia da referência para o objeto. Assim métodos executados dentro do método alteram o conteúdo do objeto compartilhado pelas duas referências.

```
Recipiente r= new Recipiente (1000);  
r.Adicionar (10);
```

1

...

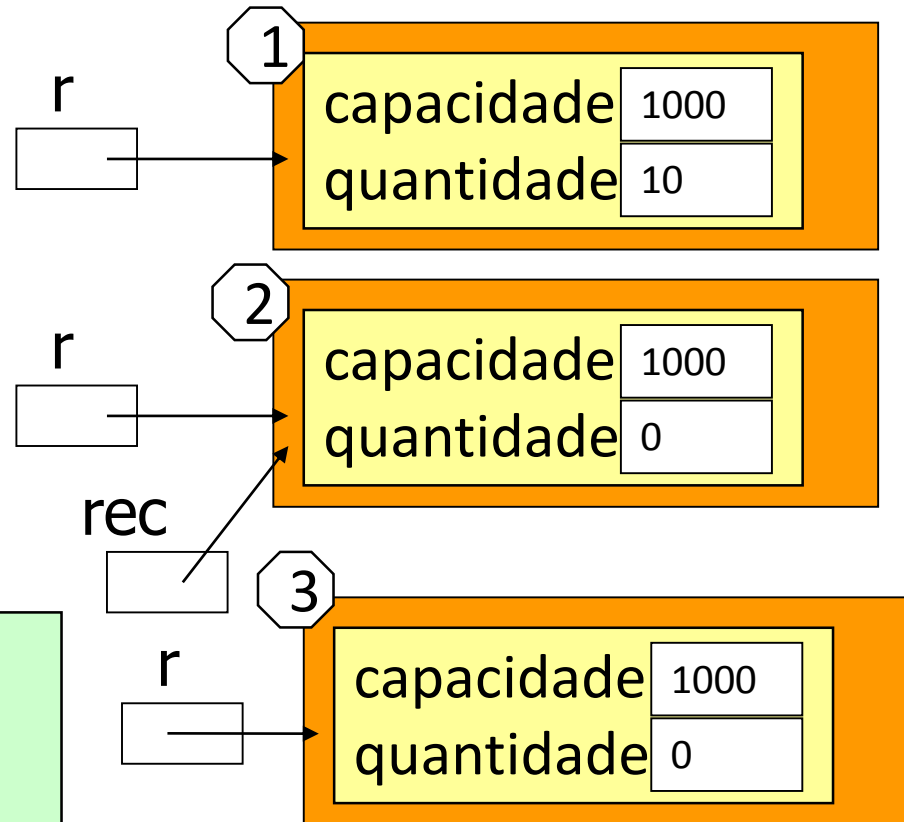
```
obj.MetodoTeste (r);
```

...

3

```
public void metodoTeste (Recipiente rec) {  
    rec.Esvaziar ( );  
}
```

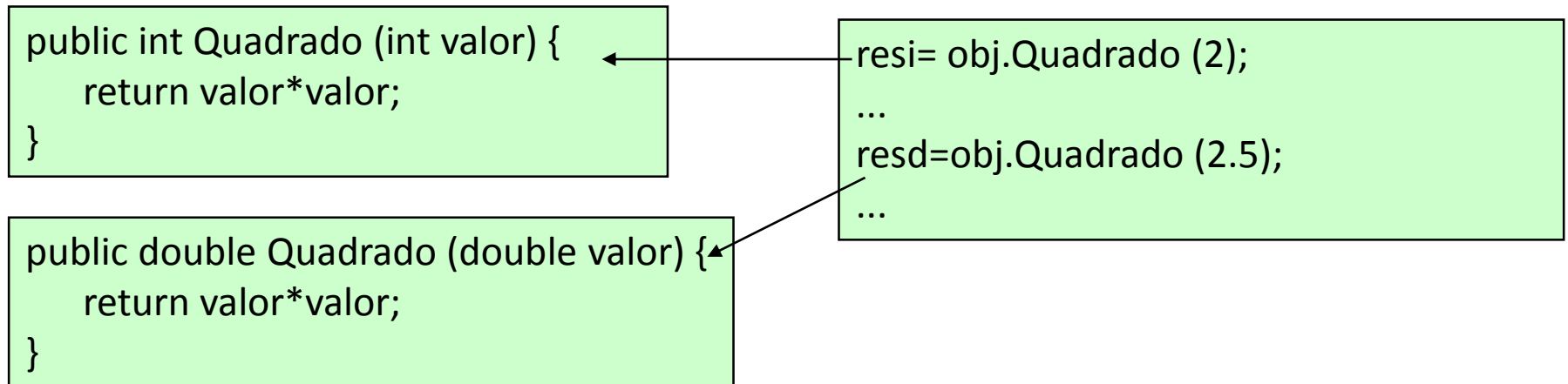
2



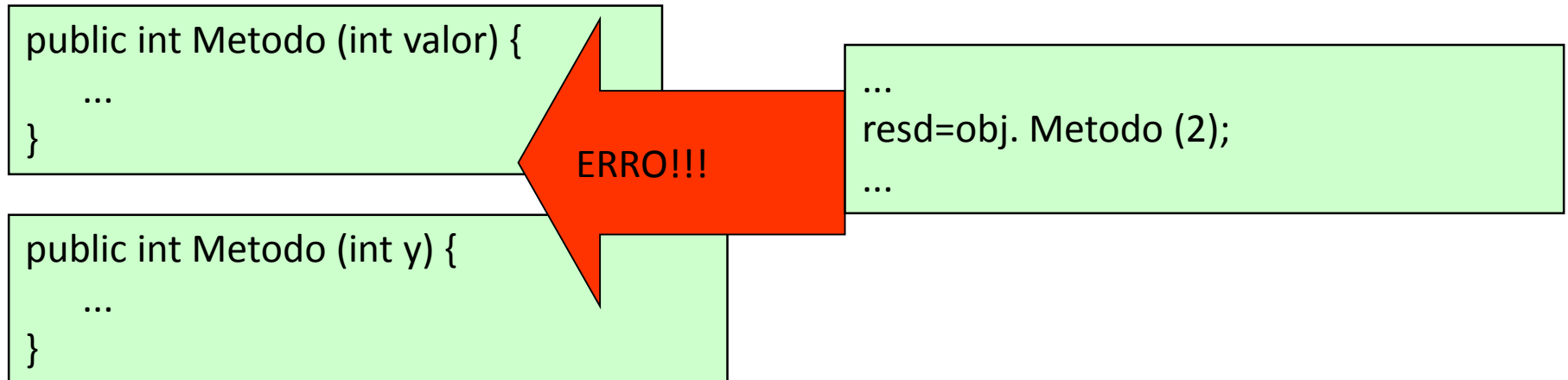
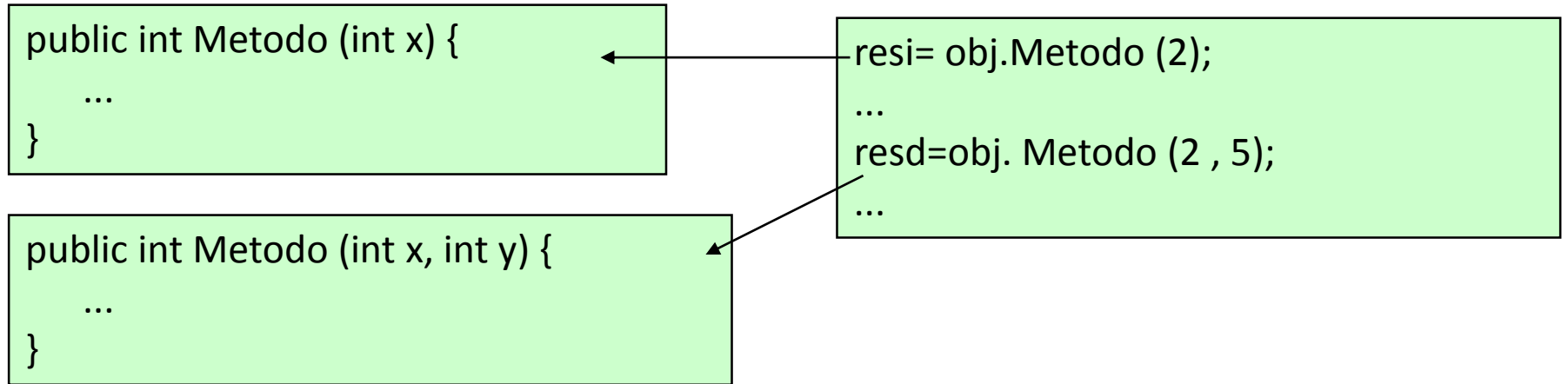
# Sobrecarga de Métodos

É possível definir para a mesma classe métodos com nomes iguais, porém com um conjunto de parâmetros diferente que possa determinar sua escolha no momento da chamada. Este recurso é chamado sobrecarga de método.

O conjunto de parâmetros pode ser diferente quanto à quantidade, ordem ou tipo.



# Sobrecarga de Métodos



Sobrecarga incorreta, pois a coleção de parâmetros não distingue os métodos. Erro em tempo de compilação.

## Padrão de Projeto: Métodos de Acesso (get/set)

São recomendações deste padrão:

- todas as variáveis de instância são declaradas com privadas e há métodos públicos para acessá-las (get/set);
- o objeto cliente pode utilizar os métodos de acesso para mudar o estado do objeto usado;
- o objeto pode acessar suas próprias variáveis de instância privadas diretamente. Porém, em alguns casos acessá-las pelos método de acesso pode contribuir com a manutenção do código.

Este tipo de padrão contribui para assegurar a consistência dos atributos, pois pode verificar valores indesejáveis por meio dos métodos de acesso.



# Padrão de Projeto: Métodos de Acesso (get/set)

```
public class Cliente {  
    private String name;  
    private boolean active;  
    ...  
    public String GetName(){  
        return name;  
    }  
    public void SetName(String name){  
        this.name=name;  
    }  
    public boolean IsActive(){  
        return active;  
    }  
    public void setActive (boolean active){  
        this.active=active;  
    }  
}
```

A palavra reservada **this** é uma referência para a própria classe.

**getXXX** e **setXXX** são nomes padrões utilizados comumente. **isXXX** é utilizado no caso de atributos booleanos.

## Atributos e métodos de classe

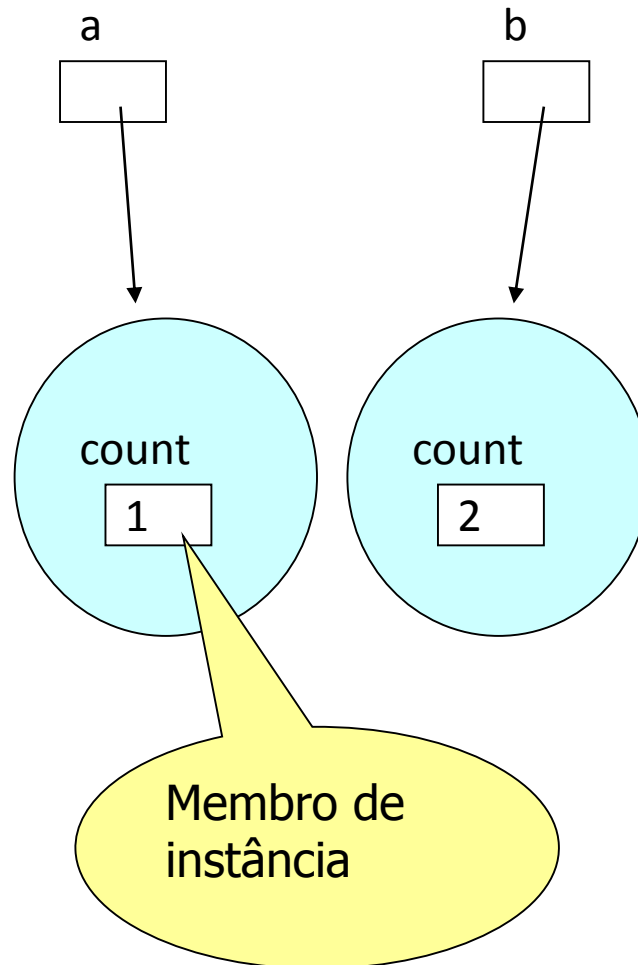
Os atributos e métodos apresentados até o momento são ditos **membros de instância**. Os atributos são alocados na memória no momento da criação da instância e os métodos atuam sobre eles durante sua existência.

É possível desenvolver atributos e métodos ditos estáticos (*static*), que ao contrário dos membros de instância são alocados no momento do carregamento da classe na memória e podem ser utilizados mesmo sem a criação de uma instância. Estes são ditos **membros de classe**.

# Atributos e métodos de classe

```
public class Counter {  
    private int count=0;  
    ...  
  
    public void Inc(){  
        count++;  
    }  
}
```

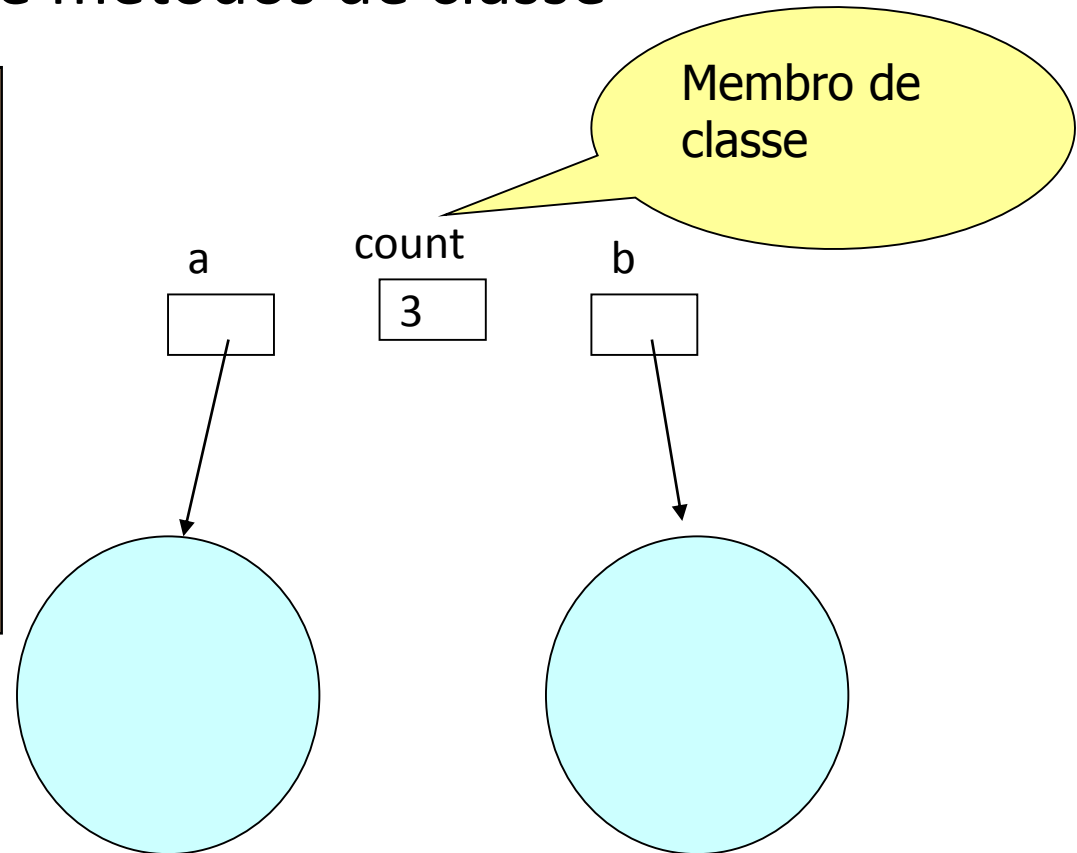
```
{  
    Counter a=new Counter ( );  
    a.Inc();  
    Counter b=new Counter( );  
    b.Inc();  
    b.Inc();  
}
```



# Atributos e métodos de classe

```
public class Counter {  
    private static int count=0;  
  
    public void Inc(){  
        count++;  
    }  
}
```

```
{  
    Counter a=new Counter ( );  
    a.Inc();  
    Counter b=new Counter( );  
    b.Inc();  
    b.Inc();  
}
```



Note que as instâncias compartilham o mesmo atributo de classe.

# Atributos e métodos de classe

```
public class Counter {  
    public static int count=0;  
  
    public void Inc(){  
        count++;  
    }  
}
```

```
{  
    ...  
    Counter.count=10;  
    ...  
}
```

count

10

Membro de  
classe

Note que se o membro de classe for público pode ser acessado utilizando o nome da classe. Não é preciso da instância de nenhum objeto.

# Atributos e métodos de classe

```
public class Counter {  
    private static int count=0;  
  
    public static void Inc(){  
        count++;  
    }  
}
```

Método  
estático

count

2

Como *inc()* é um método estático, também é possível invocá-lo por meio do nome da classe.

Um método estático pode acessar apenas outros membros de classe, não pode acessar membros de instância.

```
{  
    ...  
    Counter.Inc();  
    Counter.Inc();  
    ...  
}
```