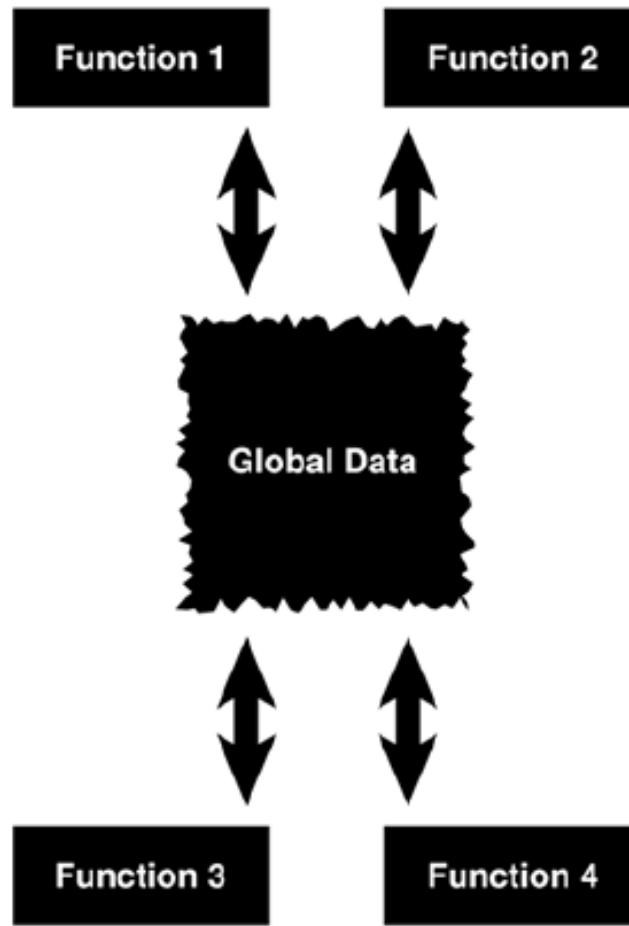


Orientação a Objetos em Java

Rodrigo Narvaes Figueira
rnfigueira@sispro.com.br

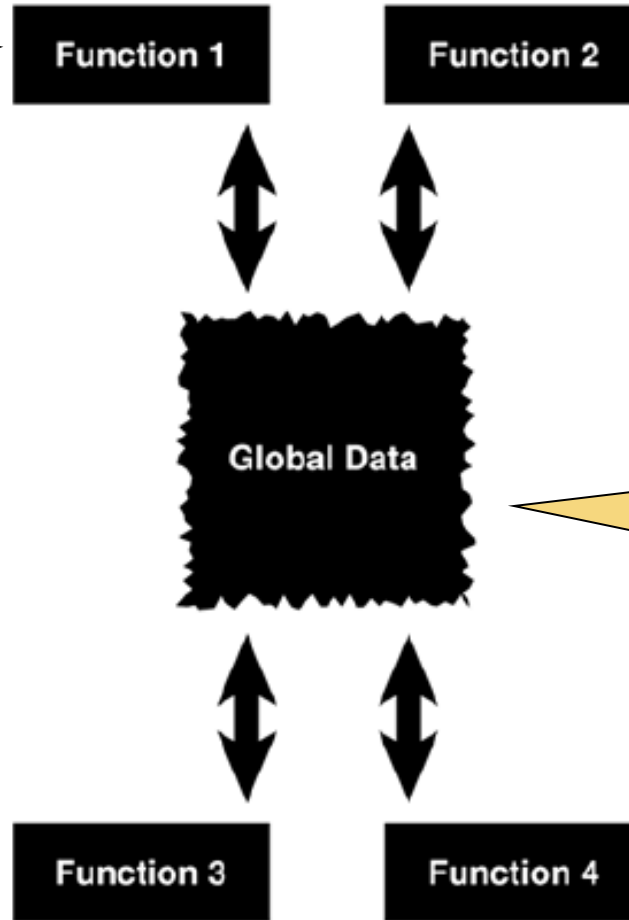
Olhando para o **passado** ...



Analizando o código ...

E se eu **modificar** esta **função**?

Irei **afetar** outras **funções**?



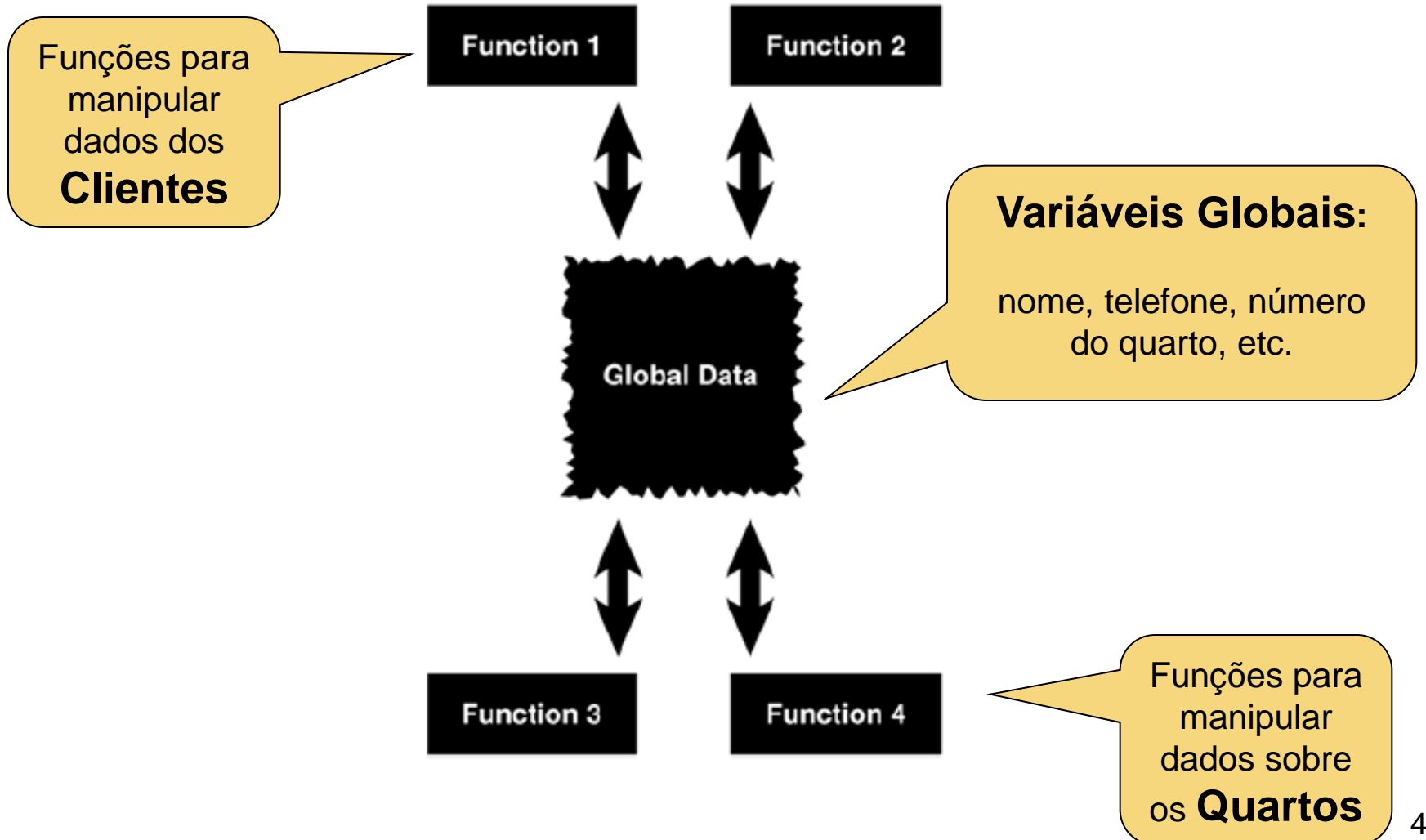
Assim, fica **difícil** eu fazer uma **modificação** neste sistema

Terei que fazer **muitos testes**

E se eu **modificar** uma **variável Global**?

Irei afetar **quantas** **funções**?

Analizando o código ...



Juntando as peças ...

- Clientes

- Funções 1 e 2
- Variáveis
 - Nome, telefone

- Quartos

- Funções 3 e 4
- Variáveis
 - Número do quarto

Juntando as peças ...

- Clientes

- Funções 1 e 2
- Variáveis
 - Nome, telefone

Somente as **funções** 1 e 2 que deveriam **manipular** as **variáveis**, nome e telefone

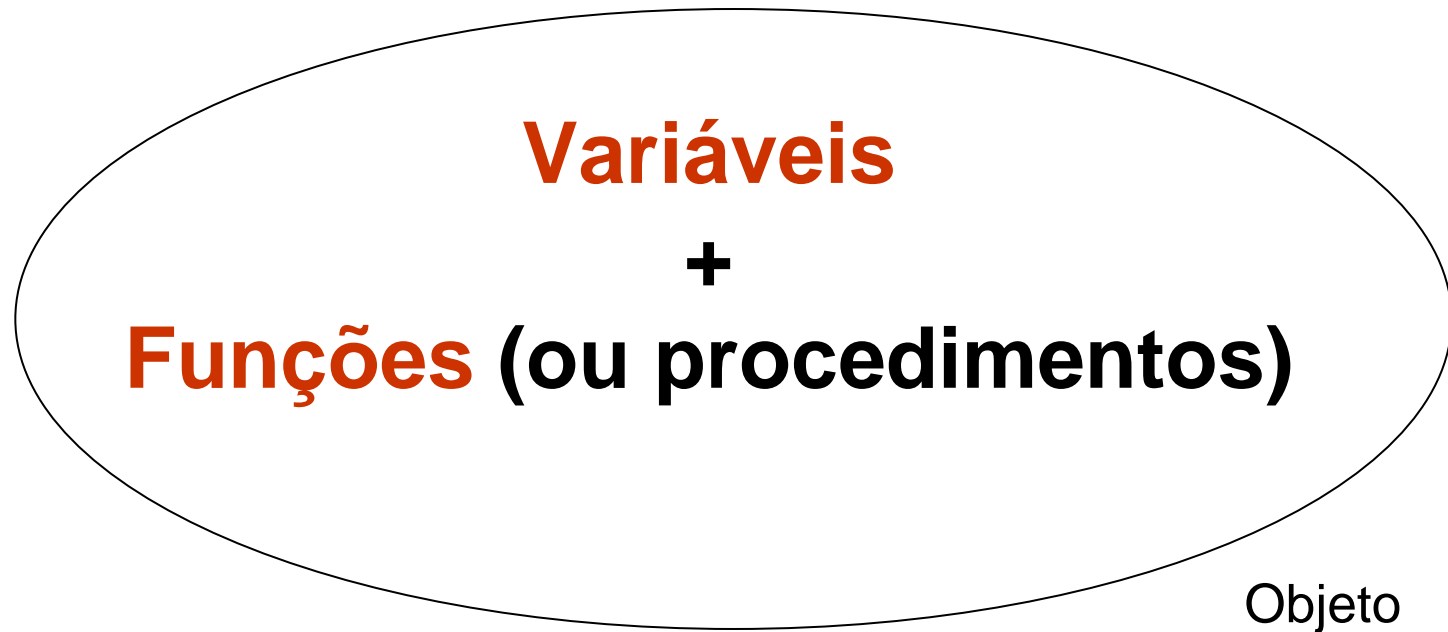
- Quartos

- Funções 3 e 4
- Variáveis
 - Número do quarto

Somente as **funções** 3 e 4 que deveriam **manipular** a **variável**, número do quarto

**Objetos são entidades que “juntam” as
“coisas”**

Entidades? Juntam?

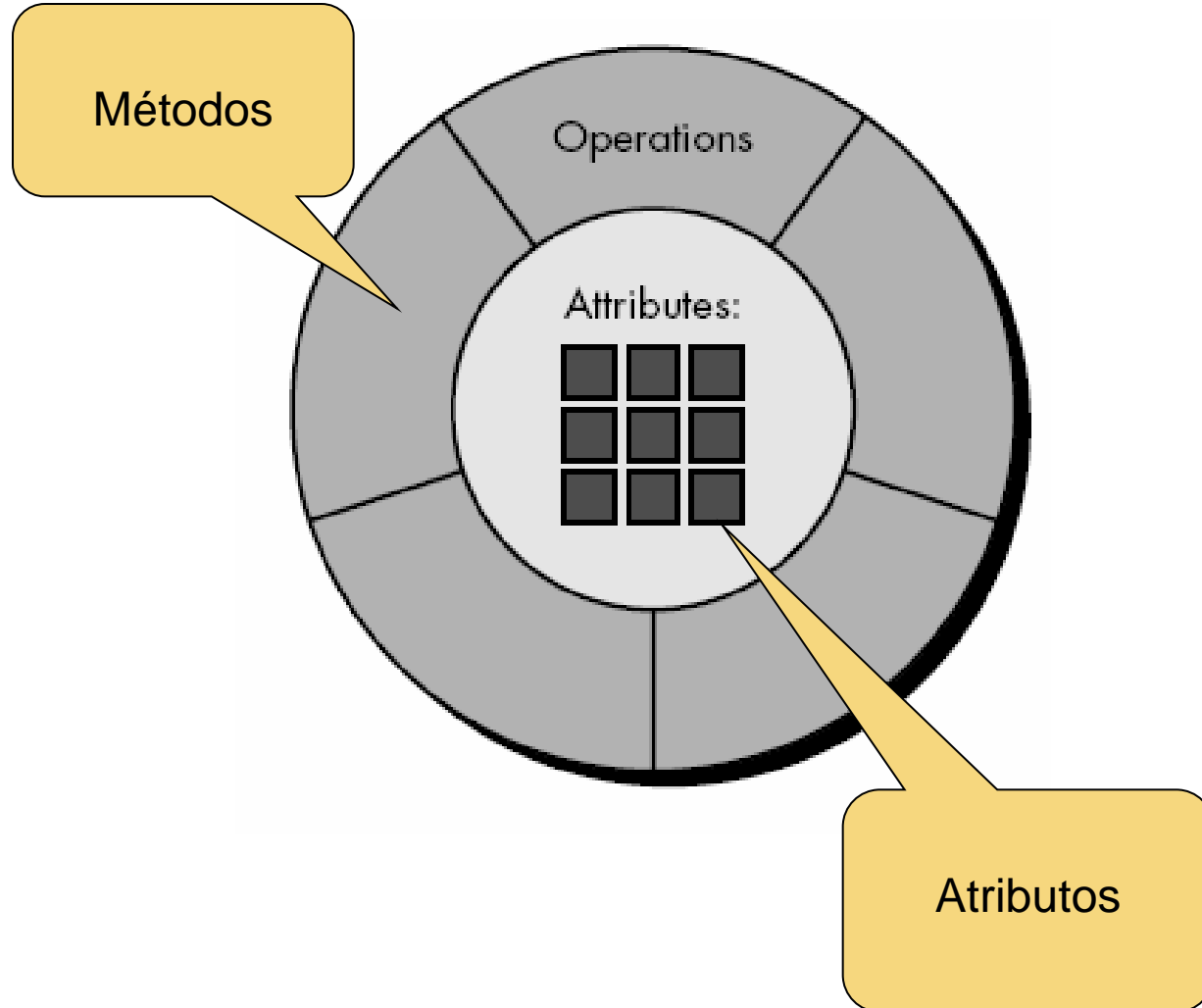


Modificando os nomes!

Variáveis \Rightarrow **Atributos**
+
Funções \Rightarrow **Métodos**

Objeto!!

Objeto = Atributos + Métodos



Objetos

Atributos definem características de um objeto

Métodos definem comportamentos ou ações que um objeto é capaz de executar

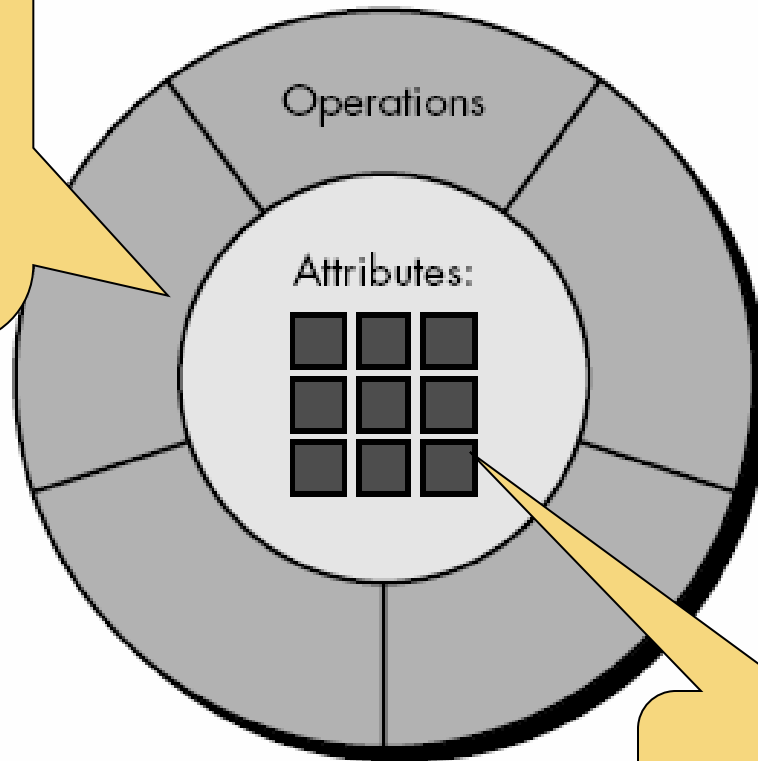
Objetos

Entidades formadas por atributos e métodos

Muitos objetos são representações do mundo real

Encapsulamento

Atributos ficam “protegidos” pelos métodos. Ou seja, somente os métodos do próprio objeto podem acessar os dados dos atributos



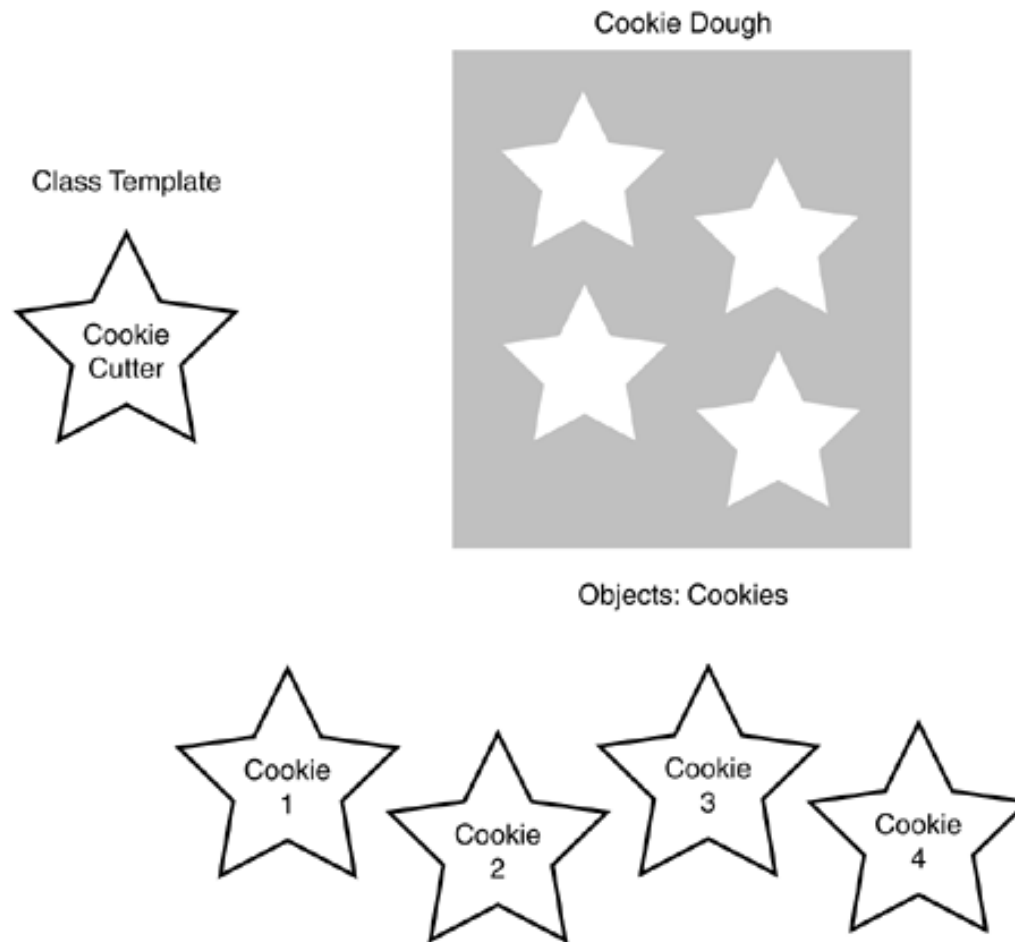
Atributos

Encapsulamento

Encapsulamento consiste em isolar aspectos internos de um objeto

E como definir e criar os meus objetos?

As **classes moldam** e **criam** os objetos



Classes

Definem o formato de um objeto
(molde ou template)

Classes instanciam (criam) os objetos

Exemplo de uma **classe** quarto em **Java**

private indica que não pode ser acessado (Encapsulado)

```
public class Quarto {
```

```
    private String numero;
```

Atributo da classe

```
    public String getNumero() {  
        return numero;  
    }
```

```
    public void setNumero(String numero) {  
        this.numero = numero;  
    }
```

```
}
```

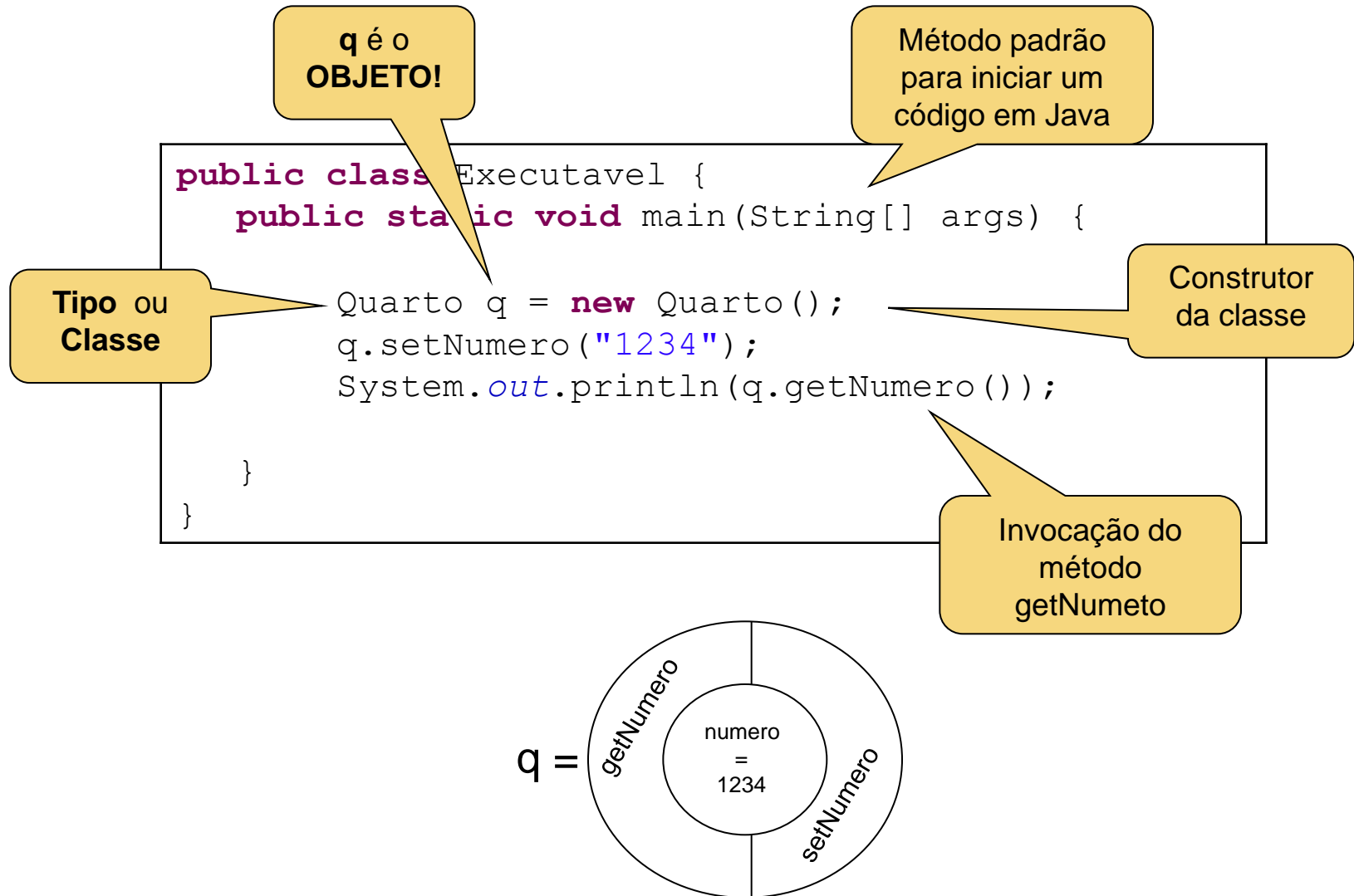
public indica que os métodos podem ser acessados (Não Encapsulados)

A classe possui **dois métodos**:

getNumero retorna o valor do atributo número

setNumero é utilizado para atribuir um valor para *número*

Instanciando (criando) um objeto em Java

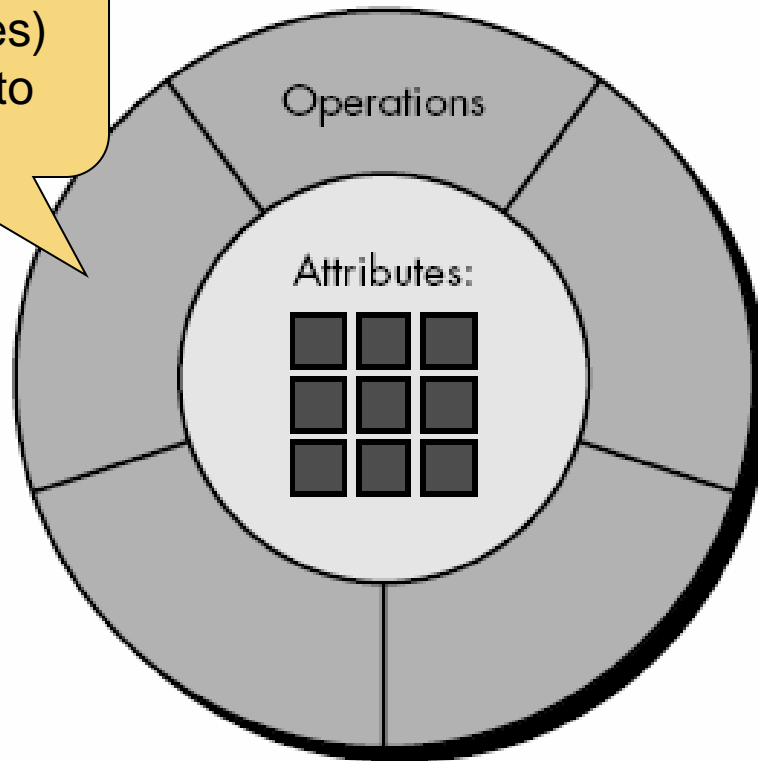


Mensagem

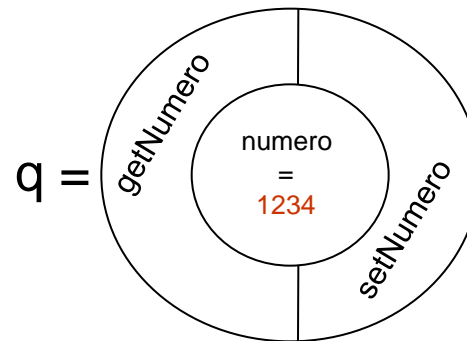
Mensagem é a invocação de um método, passando ou recebendo dados

Estado

Estados são os **dados** (valores) que um objeto carrega



Estado de q é “1234”



E se tivéssemos **clientes** pessoa **jurídica** e **física**?

Note que os **atributos** estão se repetindo

Isto implica em **mais código** para manutenção

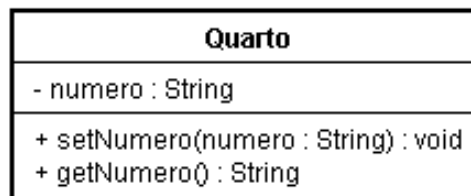
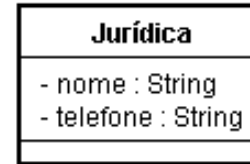
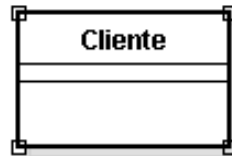
Física
- nome : String
- telefone : String

Jurídica
- nome : String
- telefone : String

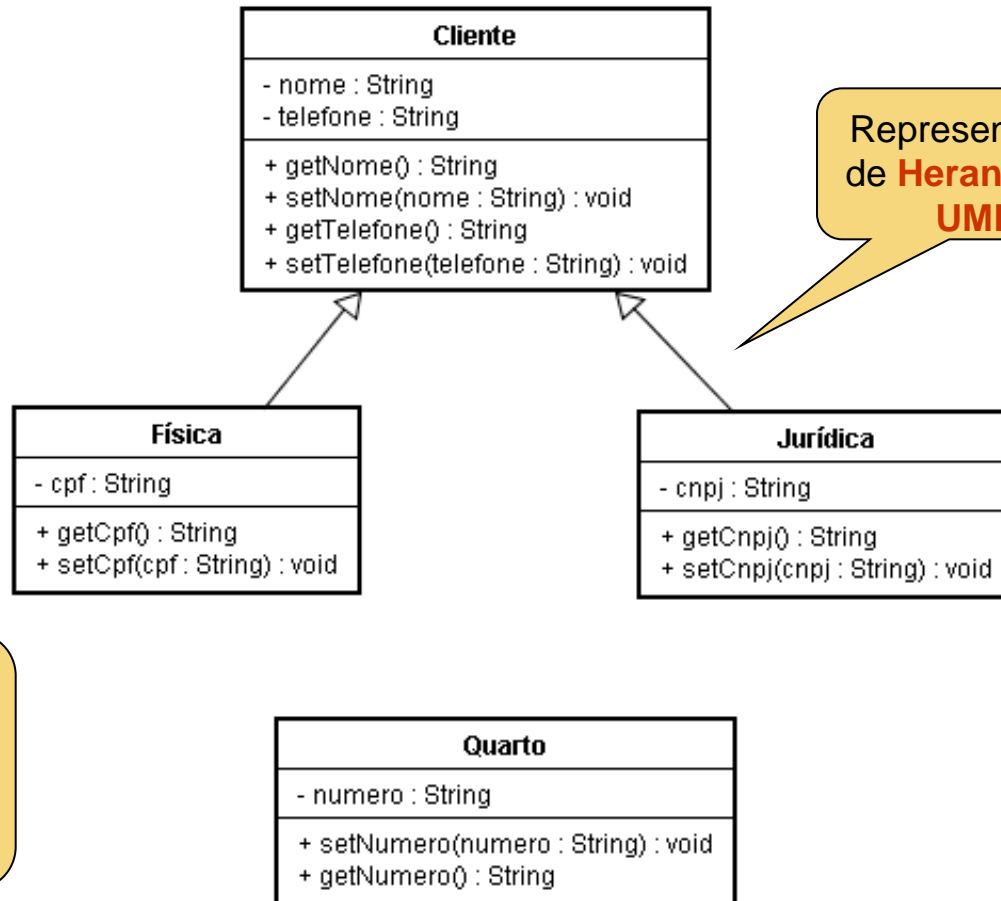
Quarto
- numero : String
+ setNumero(numero : String) : void
+ getNumero() : String

Podemos agrupar os atributos comuns numa classe genérica *Cliente*

/



As classes **Física** e **Jurídica** irão **herdar** tudo que for **público** (métodos) da classe **Cliente**



Representação
de **Herança** em
UML

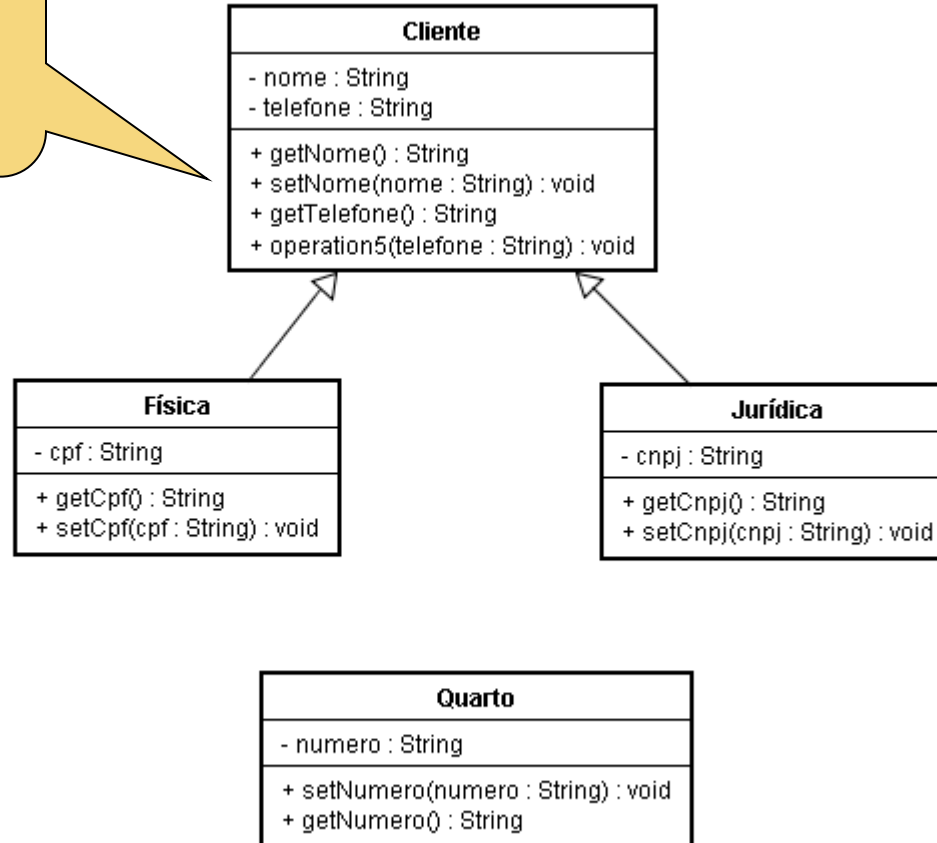
Foram encontrados
atributos específicos
(cpf e cnpj) para as
classe **Física** e
Jurídica

Nomenclatura

Generalização
OU
super classe
OU
classe pai

Especialização
OU
sub classe
OU
classe filha

Especialização
OU
sub classe
OU
classe filha



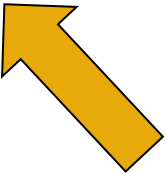
Exemplo de herança em Java

```
public class Fisica extends Cliente {  
  
    private String cpf;  
  
    public String getCpf() {  
        return cpf;  
    }  
  
    public void setCpf(String cpf) {  
        this.cpf = cpf;  
    }  
  
}
```

A palavra chave **extends** indica que a classe **Física** **herda** da classe **Cliente**

Consequentemente um **objeto f** da classe **Física** pode **invocar** o método **setNome** herdado da classe **Cliente**

```
public class Executavel {  
    public static void main(String[] args) {  
        Fisica f = new Fisica();  
  
        f.setNome("Rodrigo");  
  
        f.setCpf("4321");  
  
    }  
}
```



Herança

Herança permite que uma classe herde atributos e métodos (públicos) de outras classes

Algumas **considerações** ...

Pessoa **Física** e **Jurídica** possuem **6 métodos**

Métodos de Física: `getNome`, `setNome`, `getTelefone`, `setTelefone`
`getCpf` e `setCpf`

Métodos de Jurídica: `getNome`, `setNome`, `getTelefone`, `setTelefone`
`getCnpj` e `setCnpj`

Algumas conclusões ...

Objetos da classe **Física e Jurídica** naturalmente **são Cliente** pois **possuem** todos os **métodos** da classe **Cliente**:

getNome, setNome, getTelefone, setTelefone

Os objetos das classes Físicas e Jurídica possuem 4 métodos na sua **interface**

Interface de um Objeto


Interface é a forma de comunicação de um objeto com o meio externo

De forma objetiva, são os métodos públicos de uma classe

Podemos **obrigar** um **objeto** ter **métodos** na sua **interface**

Para isto temos que **criar** uma **obrigatoriedade**

Exemplo de Obrigatoriedade



```
public interface IFisica {  
  
    public String getCpf();  
  
    public void setCpf(String cpf);  
  
}
```

O ponto e vírgula indica que é apenas uma declaração de método, ou seja, sem corpo

O código acima irá **obrigar** que um **objeto** **tenha** determinados **métodos** em sua **interface**

A classe **Física** herda da classe **Cliente** e respeita a obrigatoriedade (*interface*) **IFísica**

```
public class Fisica extends Cliente implements IFisica {  
  
    private String cpf;  
  
    public String getCpf() {  
        return cpf;  
    }  
    public void setCpf(String cpf) {  
        this.cpf = cpf;  
    }  
}
```

A palavra chave **implements** significa que a classe respeita uma obrigatoriedade de métodos

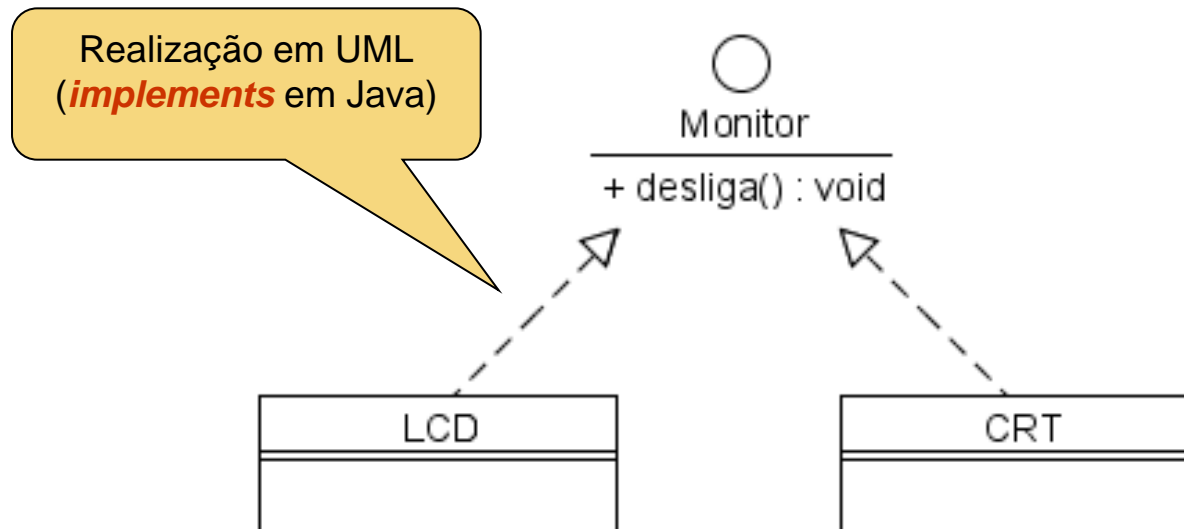
Dica 1: Uma classe em Java pode **herdar** de apenas **uma classe** e respeitar **várias obrigatoriedades** (*interfaces*)

Dica 2: Em java, um **extends** sempre deve **preceder** um **implements**

Exemplo de utilização de obrigtoriedades (*interfaces*)

o **Polimorfismo**

Notação **UML**: As classes **LCD** e **CRT** respeitam a obrigatoriedade (interface) **Monitor**



Representações em Java do slide anterior

```
public interface IMonitor {  
    public void desliga();  
}  
  
public class LCD implements IMonitor {  
    public void desliga() {  
        // Desliga um monitor de LCD  
    }  
}  
  
public class CRT implements IMonitor {  
    public void desliga() {  
        // Desliga um monitor de CRT  
    }  
}
```

Analizando o código ...

```
public class Executavel {
    public static void main(String[] args) {
        // Instanciando os objetos monitores
        LCD lcd = new LCD();
        CRT crt = new CRT();

        // Array de monitores em Java (2 posições)
        IMonitor[] array = new IMonitor[2];
        array[0] = lcd;
        array[1] = crt;

        // Desligando todos os monitores do array
        for (IMonitor monitor : array) {
            monitor.desliga();
        }
    }
}
```

Algumas Conclusões ...

O método **desliga** executou dois **códigos diferentes**

Na **primeira** volta do laço ele **desligou** um monitor de **LCD**

Na **segunda** volta do laço ele **desligou** um monitor de **CRT**

Portanto **desliga** é um método **polimórfico**
(muitas formas em tempo de execução)

Polimorfismo (Muitas Formas)

É o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse (ou que respeitem a mesma obrigatoriedade) podem invocar métodos com comportamentos distintos

Um **polimorfismo** também pode ser **implementado**
com **classes abstratas**

Classe Abstrata

Método abstrato não possui corpo e serve para realizar uma obrigatoriedade

Quando herdado, o método abstrato deve ser implementado de forma concreta

Classe Abstrata

Uma classe abstrata geralmente possui métodos concretos

Não podem ser instanciados objetos de uma classe abstrata

```

public abstract class Monitor {

    private String nome;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

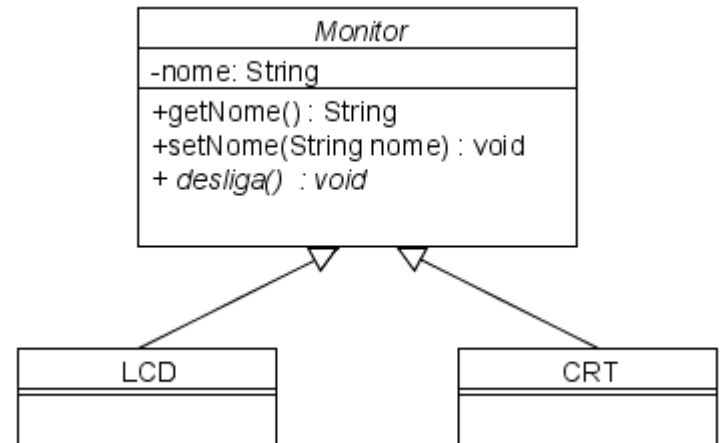
    public abstract void desliga();
}

public class LCD extends Monitor {
    public void desliga() {
        // Desliga um monitor de LCD
    }
}

public class CRT extends Monitor {
    public void desliga() {
        // Desliga um monitor de CRT
    }
}

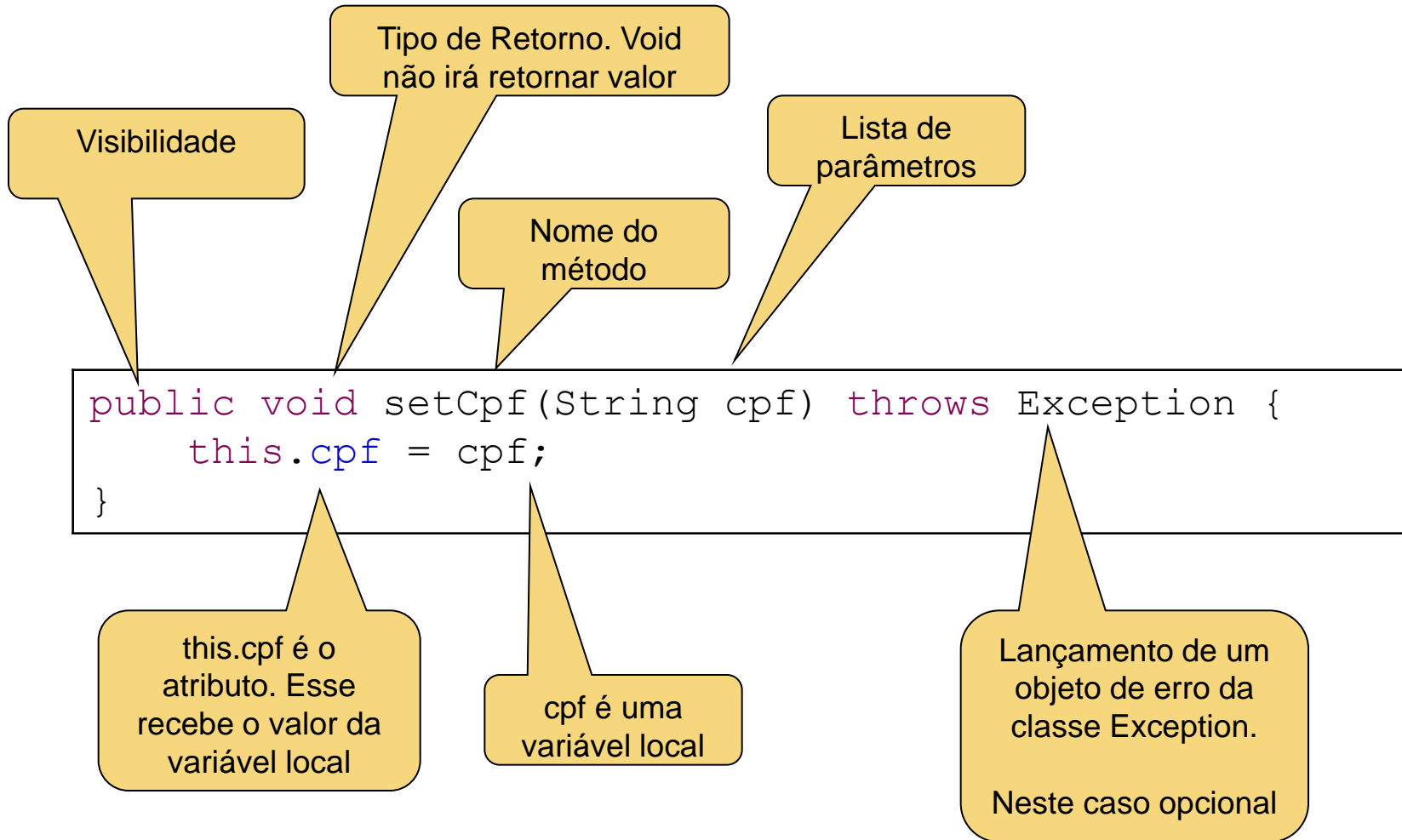
```

Exemplo Classe Abstrata



Construtores, sobrecarga e sobrescrita

Primeiro vamos **analisar** um **método** em Java



Construtor


É um método especial que serve para realizarmos a inicialização dos atributos de uma classe

Em Java um construtor não possui tipo de retorno e seu nome deve ser igual ao nome da classe

Quando não declarados, o compilador Java coloca na classe automaticamente um construtor vazio


Exemplo de construtor na classe Física

```
public class Fisica extends Cliente implements IFisica {  
  
    private String cpf;  
  
    private Fisica(){  
        this.cpf = "";  
    }  
  
    public String getCpf() {  
        return cpf;  
    }  
  
    public void setCpf(String cpf) {  
        this.cpf = cpf;  
    }  
  
}
```



Dica: Um construtor é executado apenas na criação de um objeto

```
Fisica f1 = new Fisica();  
Fisica f2 = new Fisica();
```



Construtores

- A classe *Aluno* não pode possuir um construtor *vazio* visto que o construtor da superclasse *Usuario* também não possui um construtor *vazio*
- Isto acontece pois, o construtor da superclasse sempre é executado
- Desta forma, temos que explicitamente *utilizar* a palavra chave *super* para chamar o construtor da superclasse com os argumentos necessários

```
public class Usuario
{
    private String nome;

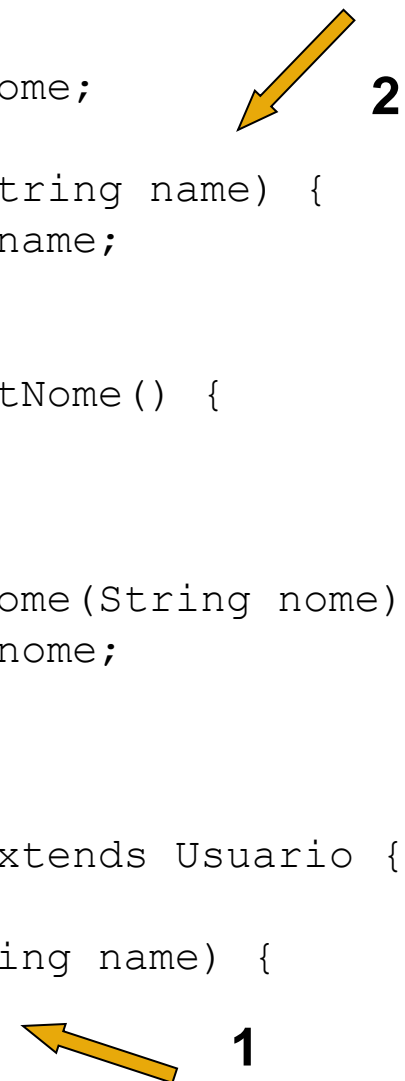
    public Usuario(String name) {
        this.nome = name;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
}

public class Aluno extends Usuario {

    public Aluno(String name) {
        super(name);
    }
}
```



2

1

Sobrecarga

Java **distingue** dois **métodos** através da **assinatura**

Assinatura de método = **nome** + **lista de parâmetros**

Assim, podemos ter **métodos** de **mesmo nome** porém
lista de **parâmetros diferente**

Analizando o código abaixo ...

```
public class Matematica {  
  
    public int soma(int a, int b) {  
        return a+b;  
    }  
  
    public int soma(int a, int b, int c) {  
        return soma(a,b) + c;  
    }  
}
```

Exemplo de
sobrecarga

A classe apresenta **métodos** de **mesmo nome** (sobrecarregados) porém
lista de **parâmetros diferentes**

Sobrecarga

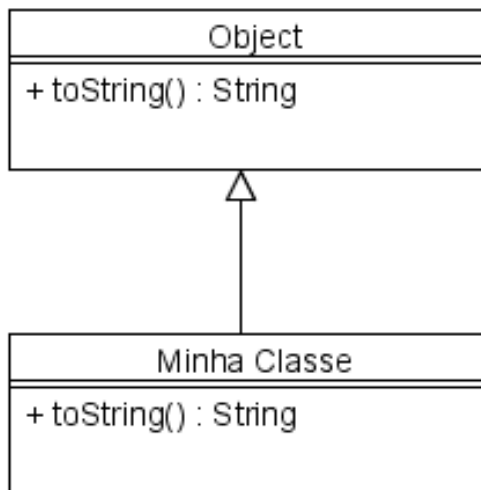
É a utilização de métodos de mesmo nome porém com funcionalidades distintas

Sobrescrita

Sobrescrita

Sobrescrita é quando uma classe desconsidera os métodos, geralmente herdados e, escreve sua própria implementação (escreve por cima)

Apesar de **herdar** o método *toString* da classe *Object*. Minha classe irá utilizar sua **própria implementação** do método *toString*



O método *toString* foi **sobrescrito** em *Minha classe*

Static

A palavra chave static em atributos

- Um objeto possui um estado.
- O estado de um objeto é dado pelos valores que atribuímos aos atributos de um objeto.
- Cada objeto possui uma área de memória particular para guardar os valores de seus atributos, ou seja, seu estado
- Em algumas situações desejamos compartilhar o valor de um atributo entre todos os objetos de uma classe

A palavra chave static em atributos

- Para isto utilizamos a palavra chave *static* para indicar que um atributo é membro da classe e não da instância
- Todos os objetos podem ler e escrever sobre os atributos estáticos, porém, seu valor é compartilhado entre todos os objetos
- Se quisermos saber, por exemplo, o número de instâncias de uma determinada classe; podemos criar um atributo estático para contar o número de objetos criados

A palavra chave static em atributos

- Exemplo de utilização de static em atributos:

```
public class Bicycle
{
    private int gear;
    private static int numberOfBicycles = 0;

    public Bicycle(int startGear){
        gear = startGear;
        numberOfBicycles++;
    }
    public static int getNumberOfBicycles() {
        return numberOfBicycles;
    }
}
```

A palavra chave static em métodos

- Por ser membro de uma classe, um método estático pode ser acessado sem instanciar um objeto:
 - `NomeDaClasse.nomeDoMetodo();`

A palavra chave static em métodos

- Você também poderá acessar o método por meio de um objeto:
 - NomeDoObjeto.nomeDoMetodo();
- Um método estático geralmente é utilizado para acessar um atributo estático:

Final

A palavra chave final

- A palavra chave final indica que um valor de um campo não pode ser modificado
- O exemplo abaixo apresenta uma constante PI definido como um atributo da classe:
 - `static final double PI = 3.141592653589793`
- Por padrão, as constantes são escritas com letra maiúscula.
- Se uma constante é composta por mais de uma palavra utiliza-se o sobrescrito (`_`) para separar as palavras.

Interfaces

Casting Implícitos e Explícitos

Casting Implícitos e Explícitos

- No exemplo abaixo podemos dizer que um objeto *MountainBike* **é** uma *Bike*, mas *Bike* **pode ser** uma *MountainBike*



- Casting mostra utilizar um objeto de um tipo no lugar de outro
- Isto somente é possível entre objetos de compartilham interfaces ou objetos da mesma hierarquia (herança).

Casting Implícitos

- Por exemplo, se escrevermos:

```
– Bike myBike = new MountainBike();
```

- O objeto *myBike* será ambos: *Bike* e *MountainBike*
- Esta operação é chamada de **Casting Implícito** e, somente é possível, pois *MountainBike* é um tipo de *Bike*

Casting Explícito

- Entretanto, se escrevermos:

- `MountainBike myMountainBike = myBike;`

- O compilador irá gerar um erro, pois, uma *Bike* nem sempre é uma *MountainBike*
- Para resolvermos este problema temos que dizer explicitamente para o compilador que o objeto *myBike* é do tipo *MountainBike* da seguinte forma:

- `MountainBike myMountainBike = (MountainBike) myBike;`

- Esta operação é chamada de *Casting* Explícito

Enum

- São tipos de campos que consistem em um conjunto fixo de constantes (static final), sendo como uma lista de valores pré-definidos

```
public enum Cartas {  
    A, J, Q, K;  
}
```

Referências Bibliográficas

- Weisfeld, M.; **Object-Oriented Thought Process**. Editora: Sams, ISBN-10: 0672326116.
- Pressman, R.; **Software Engineering: A Practitioner's**. McGraw-Hill Science Approach ISBN-10: 007301933X
- <http://docs.oracle.com/javase/tutorial/>