Programação Orientada por Objetos

Herança

Prof. Cédric Grueau
Prof. José Sena Pereira
Departamento de Sistemas e Informática
Escola Superior de Tecnologia de Setúbal
Instituto Politécnico de Setúbal
2022/2023

Sumário

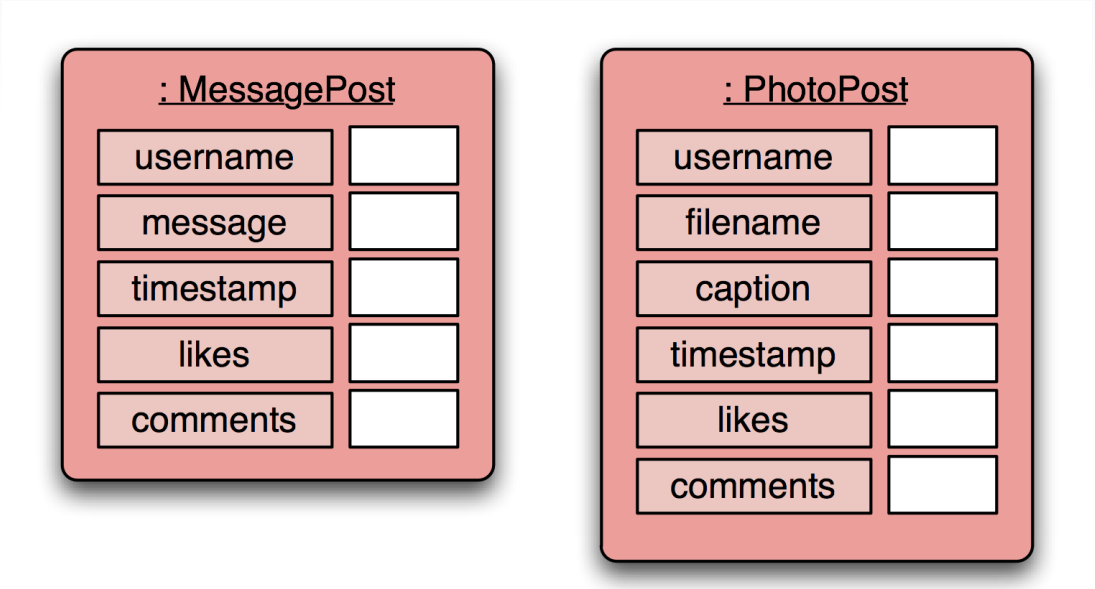
- Herança de classes
- Hierarquias de classes
- Principio da substituição

Exemplo – Rede Social

- Requisitos da rede social:
  - Um pequeno protótipo com a base para o armazenamento e apresentação de mensagens.
  - Faz o armazenamento de mensagens de texto e mensagens de imagem.
    - As mensagens de texto podem ter várias linhas;
    - as mensagens de imagem têm uma imagem e uma descrição.
  - Todas as mensagens devem incluir o seu autor, a altura em que foi enviada, o número de "gostos" e a lista de comentários.

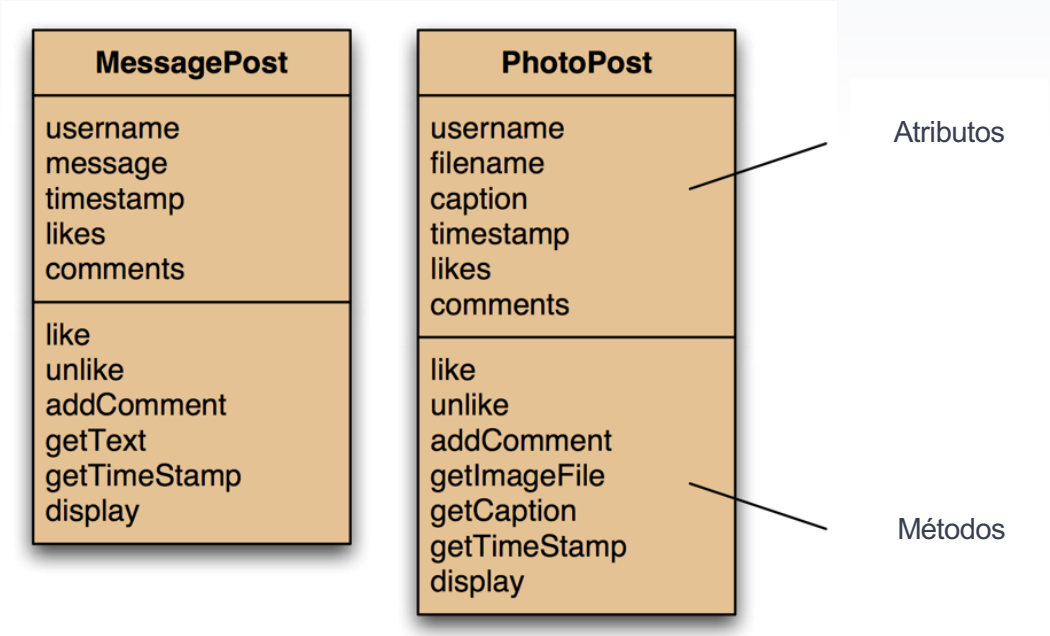
Exemplo – Rede Social

- Objetos que pretendemos representar: **`MessagePost`** e **`PhotoPost`**



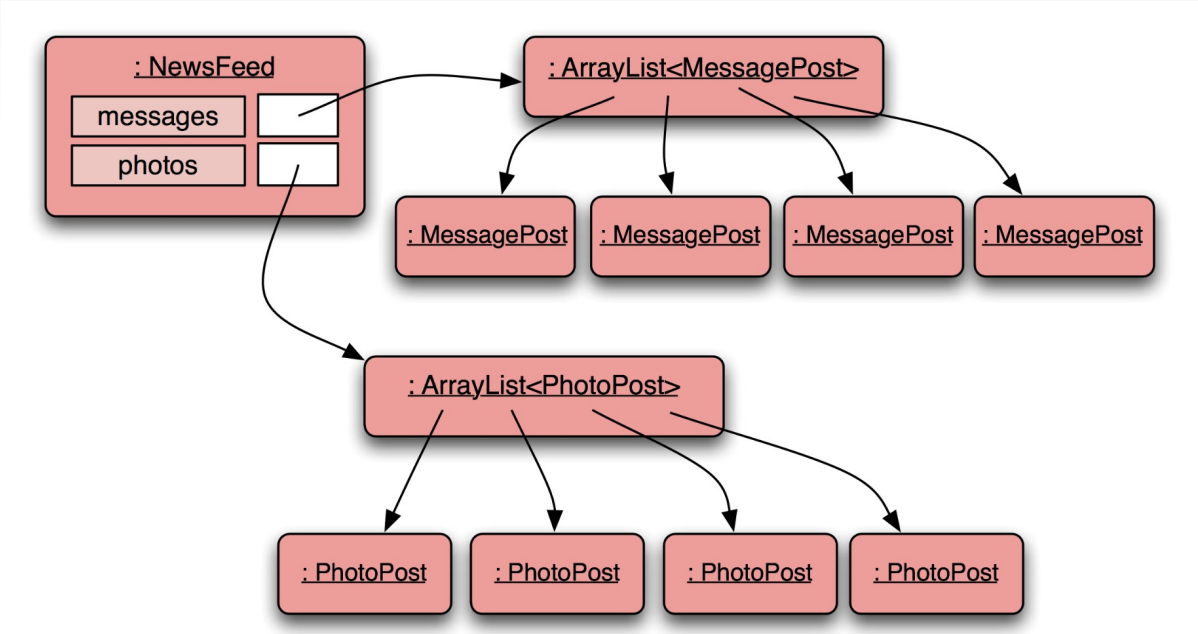
Exemplo – Rede Social

- Classes associadas: **`MessagePost`** e **`PhotoPost`**



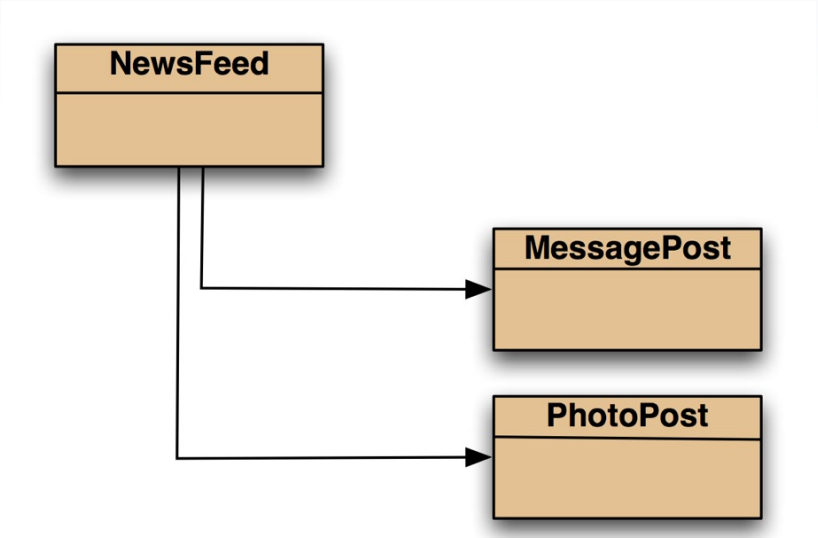
Exemplo – Rede Social

- **Diagrama de objetos** da rede social



Exemplo – Rede Social

- Diagrama de classes da rede social



Exemplo – Rede Social

- Classe **MessagePost**

```
public class MessagePost {

    private String username;
    private String message;
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    public MessagePost(String author, String text) {
        username = author;
        message = text;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<String>();
    }

// continua…
```

(System.currentTimeMillis – das bibliotecas do Java)

currentTimeMillis

```
public static long currentTimeMillis()
```

Returns the current time in milliseconds. Note that while the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger. For example, many operating systems measure time in units of tens of milliseconds.

See the description of the class `Date` for a discussion of slight discrepancies that may arise between "computer time" and coordinated universal time (UTC).

Returns:

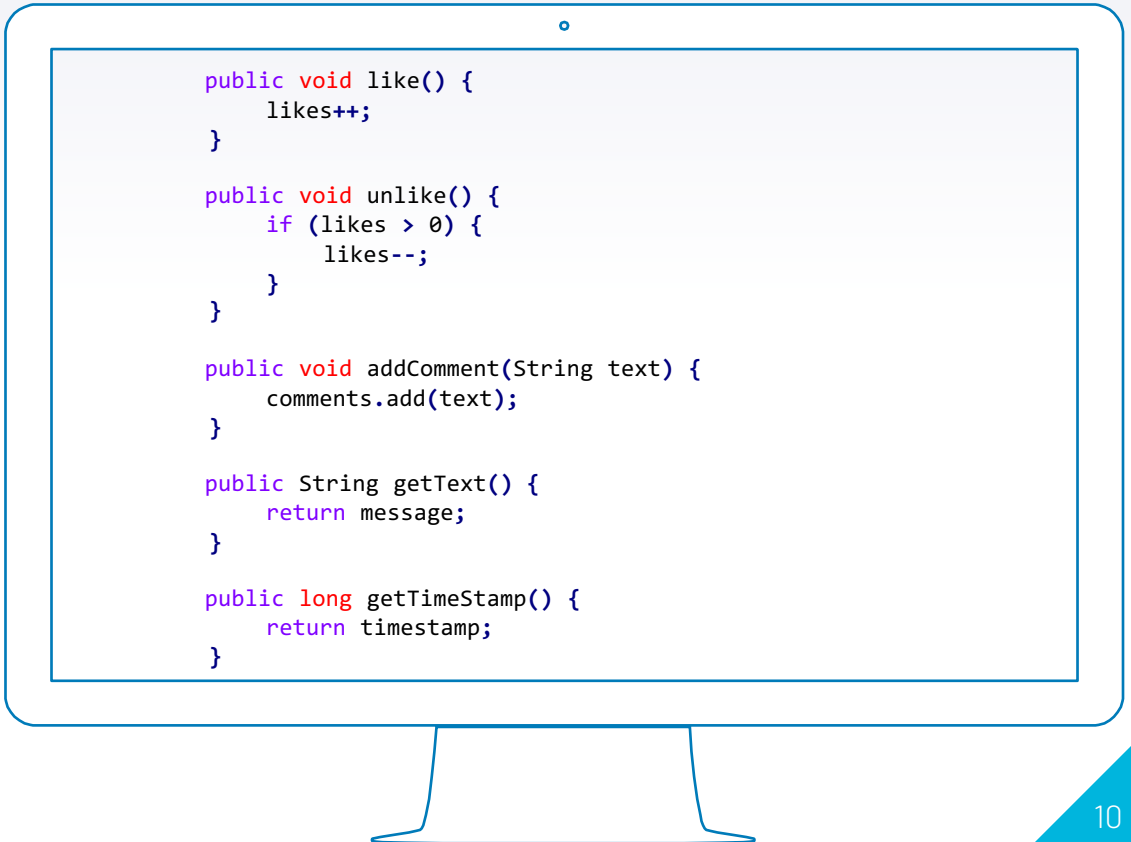
the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.

See Also:

`Date`

Exemplo – Rede Social

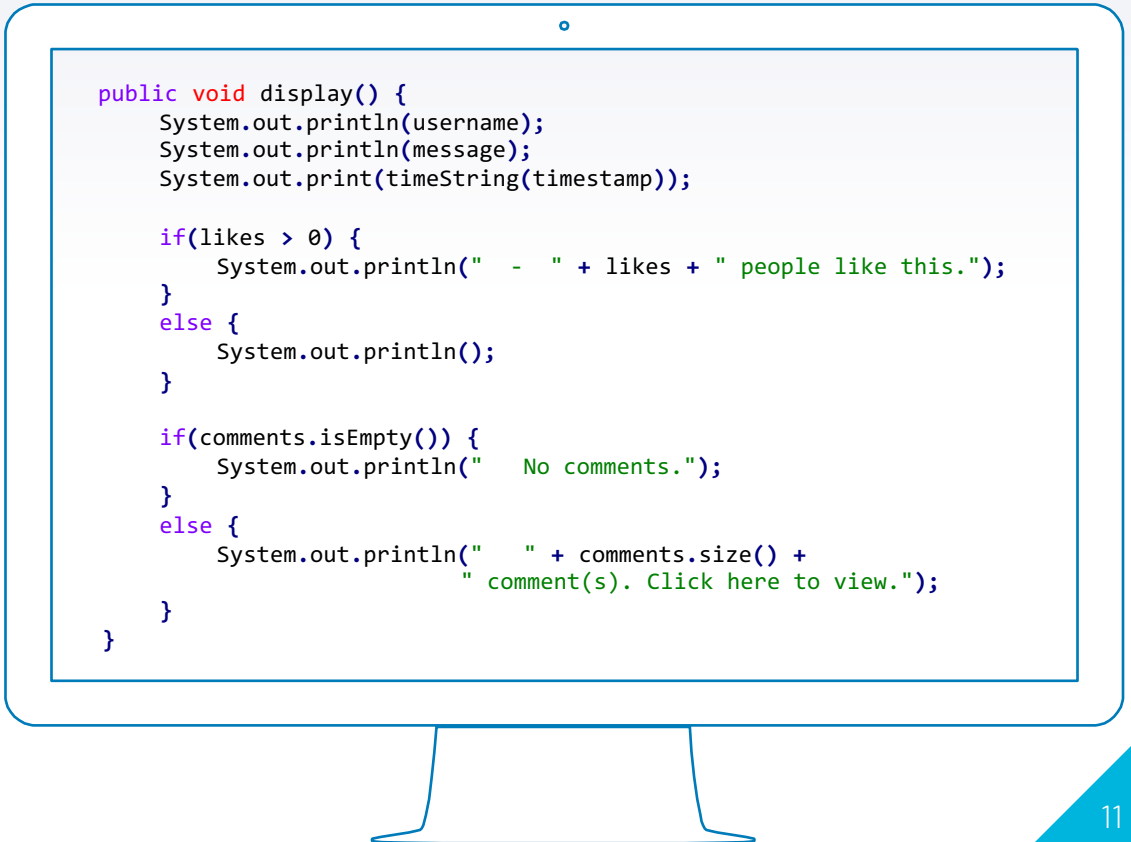
- Métodos da classe **MessagePost** (1/3)



```
public void like() {
    likes++;
}

public void unlike() {
    if (likes > 0) {
        likes--;
    }
}

public void addComment(String text) {
    comments.add(text);
}

public String getText() {
    return message;
}

public long getTimeStamp() {
    return timestamp;
}
```

Exemplo – Rede Social

- Métodos da classe **MessagePost** (2/3)



```
public void display() {
    System.out.println(username);
    System.out.println(message);
    System.out.print(timeString(timestamp));

    if(likes > 0) {
        System.out.println("  -  " + likes + " people like this.");
    }
    else {
        System.out.println();
    }

    if(comments.isEmpty()) {
        System.out.println("   No comments.");
    }
    else {
        System.out.println("   " + comments.size() +
                           " comment(s). Click here to view.");
    }
}
```

Exemplo – Rede Social

- Métodos da classe **MessagePost** (3/3)

Tempo passado em milissegundos

```
private String timeString(long time) {
    long current = System.currentTimeMillis();
    long pastMillis = current - time;
    long seconds = pastMillis/1000;
    long minutes = seconds/60;
    if(minutes > 0) {
        return minutes + " minutes ago";
    }
    else {
        return seconds + " seconds ago";
    }
}
```

Exemplo – Rede Social

```
public class PhotoPost {
     private String username;
     private String filename;
     private String caption;
     private long timestamp;
     private int likes;
     private ArrayList<String> comments;

    public PhotoPost(String author, String filename, String caption)
     {
          username = author;
          this.filename = filename;
          this.caption = caption;
          timestamp = System.currentTimeMillis();
          likes = 0;
          comments = new ArrayList<String>();
     }

// continua…
```

Nome do ficheiro com a imagem

Legenda da imagem

- Classe **PhotoPost**

Exemplo – Rede Social

```
public void like() {
    likes++;
}

public void unlike() {
    if (likes > 0) {
        likes--;
    }
}

public void addComment(String text) {
    comments.add(text);
}

public String getImageFile()  {
    return filename;
}

public String getCaption() {
    return caption;
}
```

- Métodos da classe **PhotoPost** (1/3)

Exemplo – Rede Social

```
public long getTimeStamp() {
    return timestamp;
}

public void display() {
    System.out.println(username);
    System.out.println("  [" + filename + "]");
    System.out.println("  " + caption);
    System.out.print(timeString(timestamp));
    if(likes > 0) {
        System.out.println("  -  " + likes + " people like this.");
    }
    else {
        System.out.println();
    }
    if(comments.isEmpty()) {
        System.out.println("   No comments.");
    }
    else {
        System.out.println("   " + comments.size() + " comment(s). Click here to view.");
    }
}
```

- Métodos da classe **PhotoPost** (2/3)

Exemplo – Rede Social

```
private String timeString(long time) {
    long current = System.currentTimeMillis();
    long pastMillis = current - time;
    long seconds = pastMillis/1000;
    long minutes = seconds/60;
    if(minutes > 0) {
        return minutes + " minutes ago";
    }
    else {
        return seconds + " seconds ago";
    }
}
```

- Métodos da classe **PhotoPost** (3/3)

Exemplo – Rede Social

```
public class NewsFeed  {
    private ArrayList<MessagePost> messages;
    private ArrayList<PhotoPost> photos;

    public NewsFeed() {
        messages = new ArrayList<MessagePost>();
        photos = new ArrayList<PhotoPost>();
    }

    public void addMessagePost(MessagePost message) {
        messages.add(message);
    }

    public void addPhotoPost(PhotoPost photo) {
        photos.add(photo);
    }

     // continua…
```

- Classe **NewsFeed**

Exemplo – Rede Social

```
    // continuação da classe Newsfeed…

    public void show() {
        // display all text posts
        for(MessagePost message : messages) {
            message.display();
            System.out.println(); // empty line between posts
        }

        // display all photos
        for(PhotoPost photo : photos) {
            photo.display();
            System.out.println(); // empty line between posts
        }
    }
}
```

- Classe **NewsFeed**

Exemplo – Rede Social

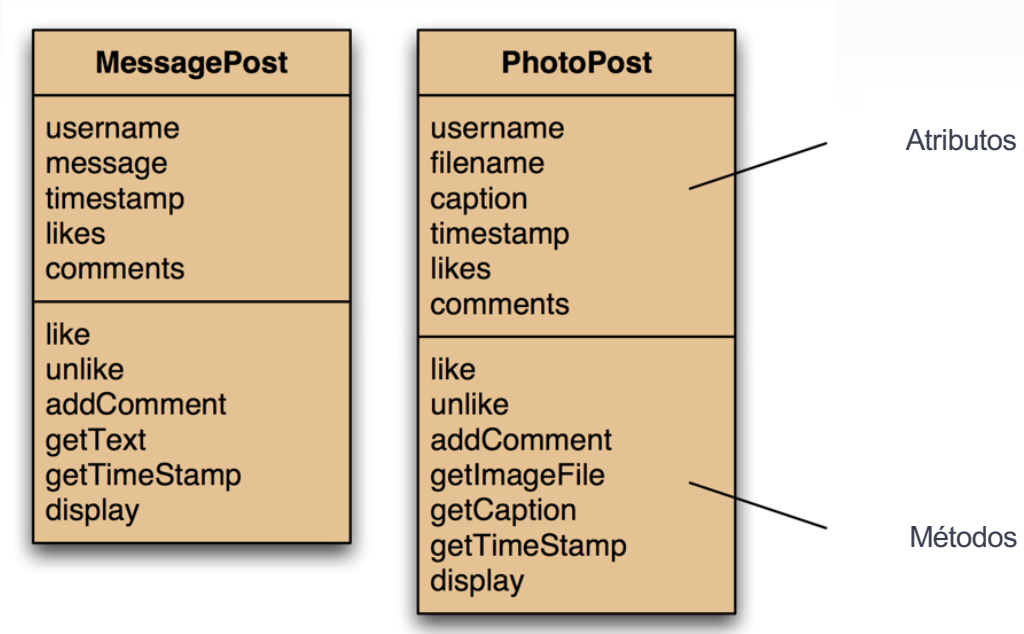
- **Problemas do protótipo** criado:
  - Duplicação de código
    - As classes **`MessagePost`** e **`PhotoPost`** são bastante parecidas (grande parte do código é idêntico).
    - A manutenção do código dá mais trabalho.
    - Corre-se o risco de se criarem *bugs* se não se alterar em todos os locais onde o código está em duplicado.
    - A classe **`NewsFeed`** também tem duplicação de código.

Herança de Classes

- Herança de Classes

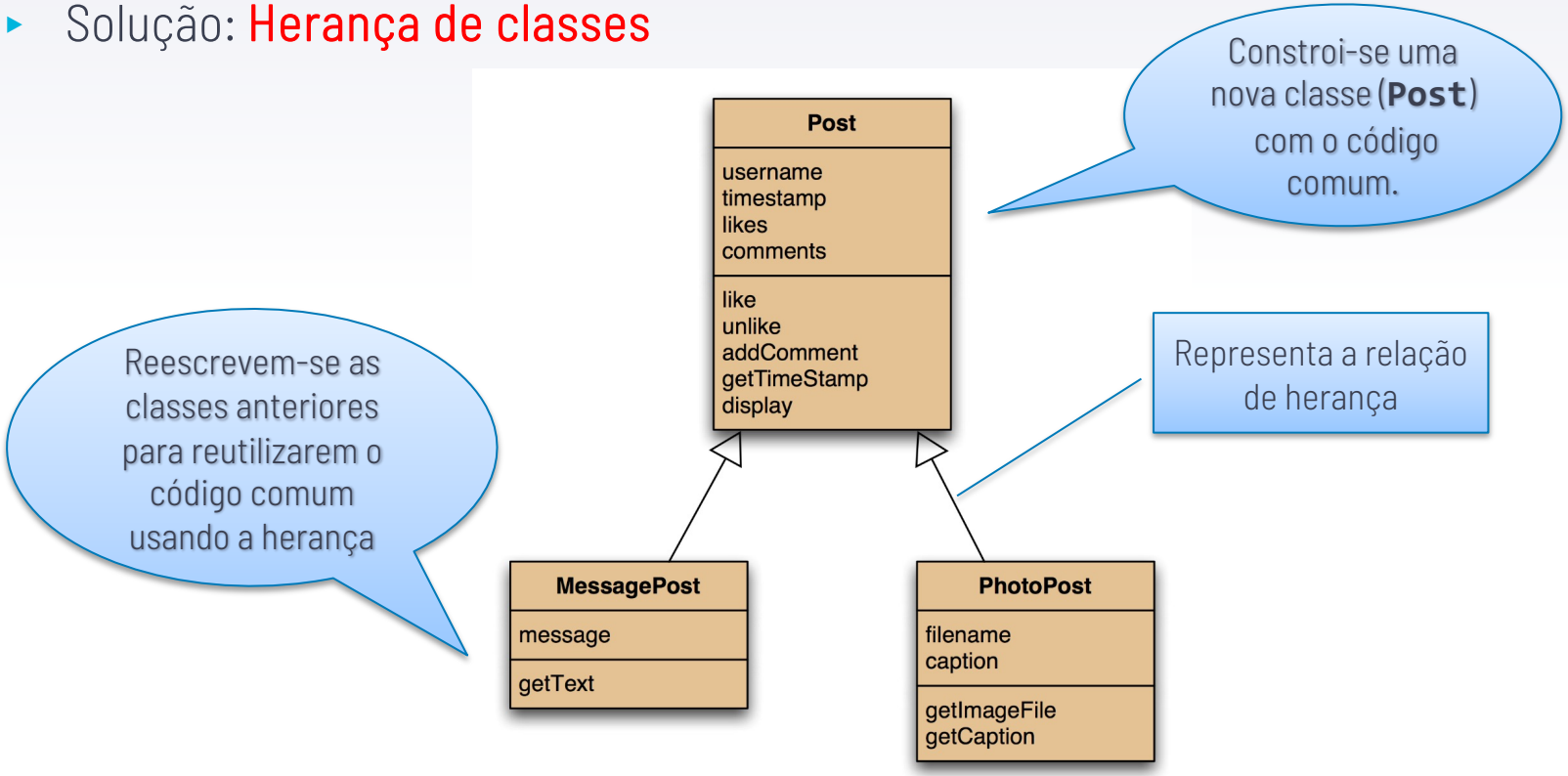
Exemplo – Rede Social

- Classes **`MessagePost`** e **`PhotoPost`**
  - Problema: Código duplicado



Exemplo – Rede Social

- Solução: Herança de classes



Herança de classes

Herança de classes

- É uma técnica usada em Programação Orientada por Objetos que vai permitir a reutilização de código.
  - Define-se uma classe com o código comum.
  - Criam-se outras classes com base na classe anterior que reutilizam esse código.
  - A classe criada inicialmente é a **superclasse** e as classes que vão reutilizar essa classe são as **subclasses**.
  - Ao processo de reutilização de uma classe dá-se o nome de **Herança**
  - As subclasses vão **herdar** o código da superclasse

Exemplo – Rede Social

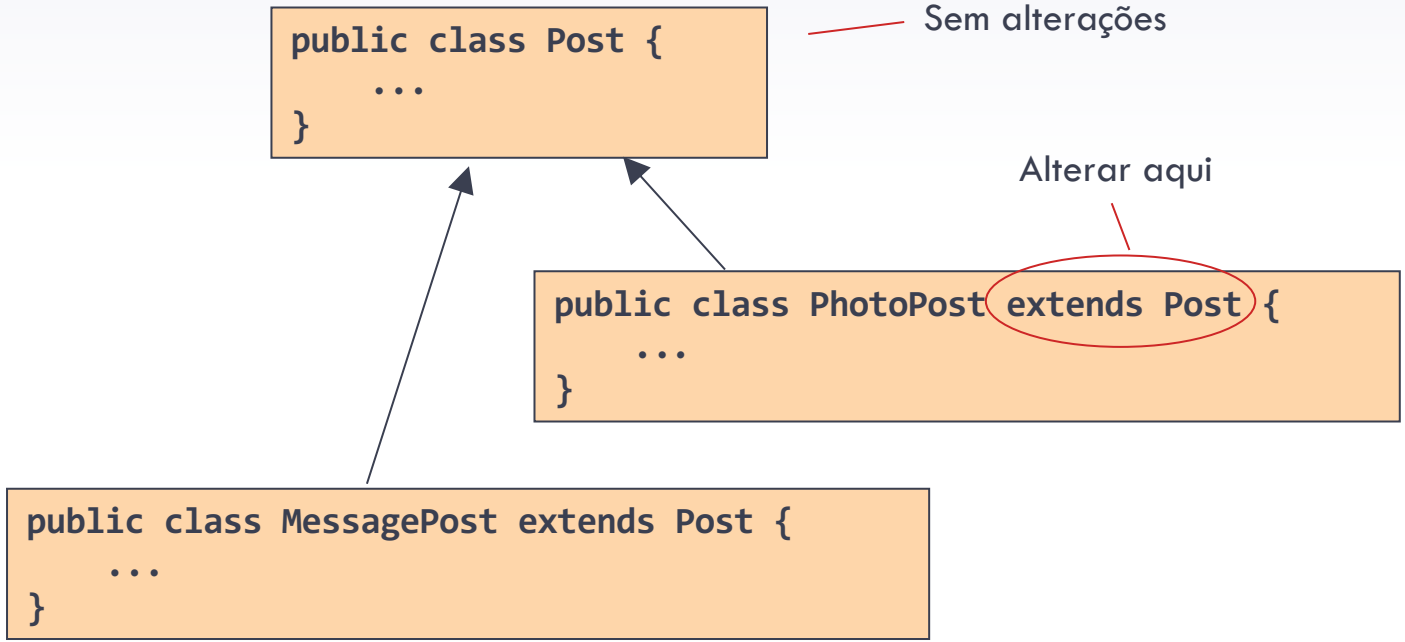
- Solução: usar a Herança de classes

Herança de classes (receita simples)

1. Define-se uma superclasse : **`Post`**
2. Define-se subclasses para **`MessagePost`** e **`PhotoPost`**
3. Na superclasse definem-se os atributos comuns
4. As subclasses herdam os atributos da superclasse
5. As subclasses adicionam outros atributos

Exemplo – Rede Social

- Herança de classes em Java



Exemplo – Rede Social

```
public class Post {
    private String username;
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    // construtores e métodos omitidos
}
```

- Superclasse – **Post**

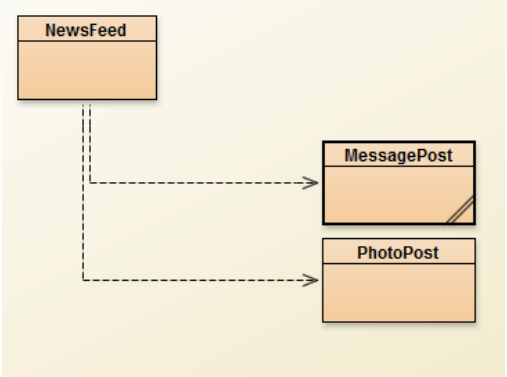
Exemplo – Rede Social

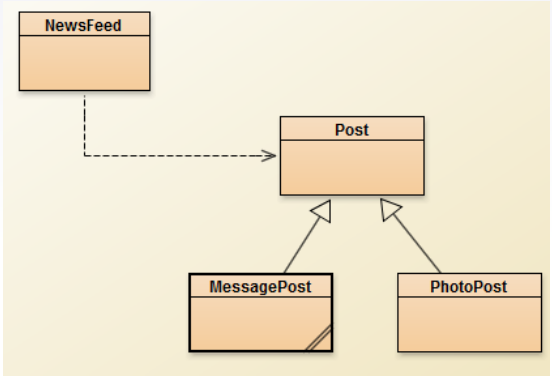
```
public class MessagePost extends Post {
    private String message;

    // construtores e métodos omitidos
}

public class PhotoPost extends Post {
    private String filename;
    private String caption;

    // construtores e métodos omitidos
}
```

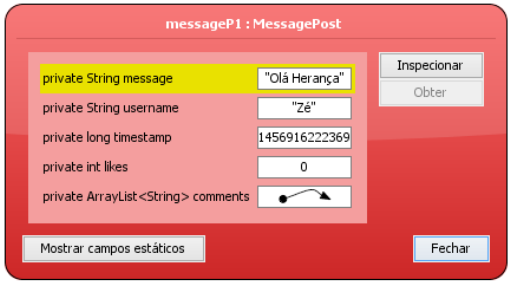
- Subclasses – **MessagePost** e **PhotoPost**

Exemplo – Rede Social

- Herança de classes em Java

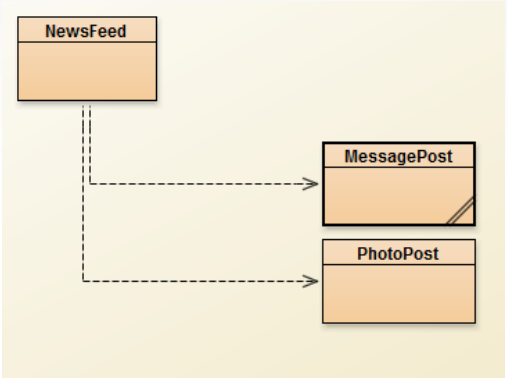


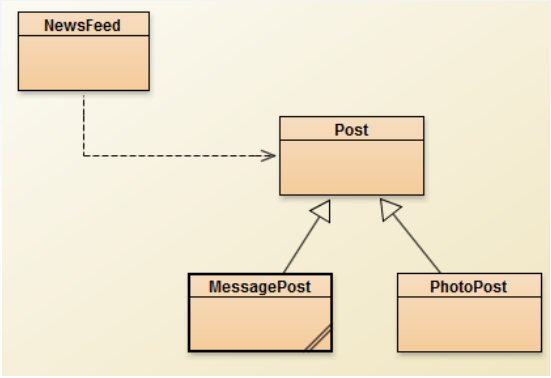


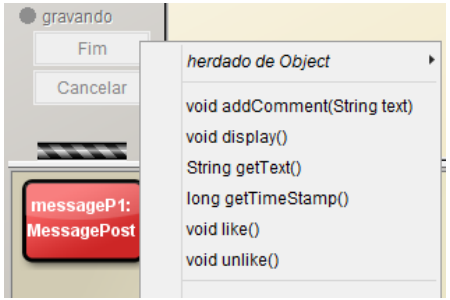


Exemplo – Rede Social

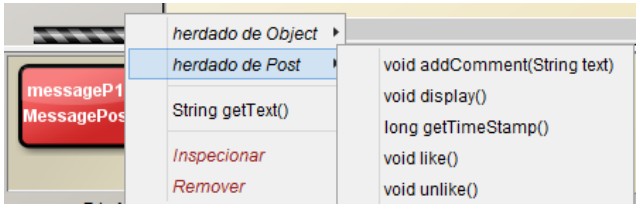
- Herança de classes em Java







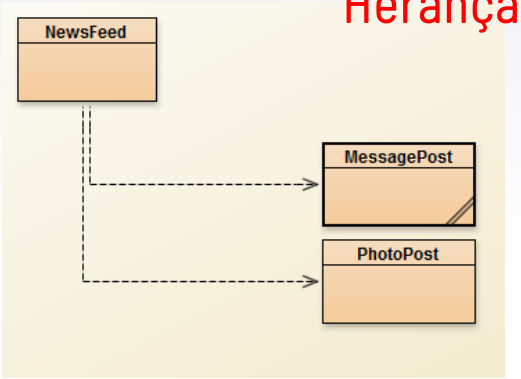
Métodos idênticos

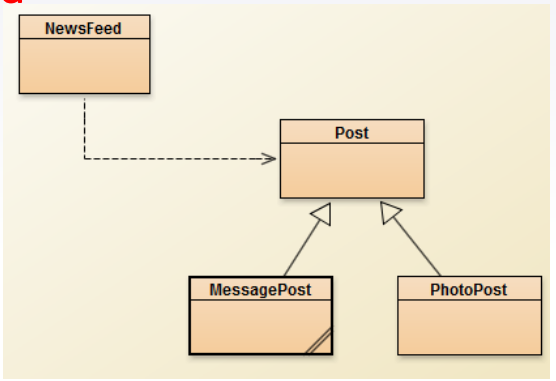


O BlueJ agrupa os métodos nas classes a que pertencem no menu mas a utilização em código não tem diferenças.

Exemplo – Rede Social

Herança de classes em Java

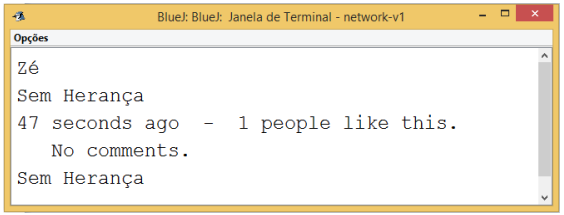


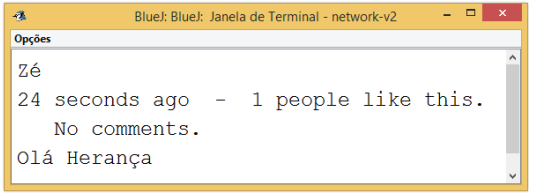



```
MessagePost post = new MessagePost("Zé", "Sem Herança");

post.like();
post.display();
String text = post.getText();
System.out.println(text);
```

Mesma utilização

```
MessagePost post = new MessagePost("Zé", "Olá Herança");

post.like();
post.display();
String text = post.getText();
System.out.println(text);
```





Herança de classes

- Herança de classes
  - A herança de classe define uma relação "is-a" entre classes.
    - Ex: A classe **`MessagePost`** herda da classe **`Post`**:

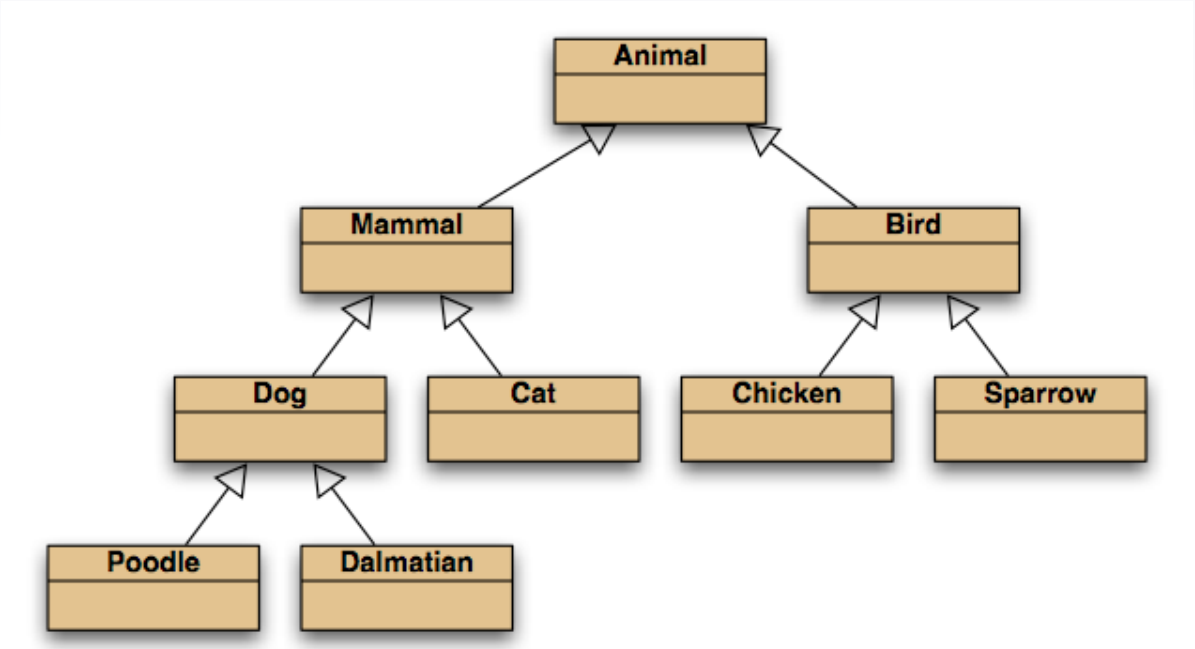
      Uma mensagem de texto (**`MessagePost`**) é uma (Is-a) mensagem (**`Post`**)
    - Ex: A classe **`Carro`** herda da classe **`Veiculo`**:

      Um carro é um (Is-a) veiculo
  - Enquanto a composição define uma relação "has-a" entre classes
    - Ex: A classe **`Carro`** contém (has-a) um objeto da classe **`Motor`**:

      Um carro contém (has-a) motor

Hierarquias de classes

- A herança de classes leva à formação de **hierarquias de classes**

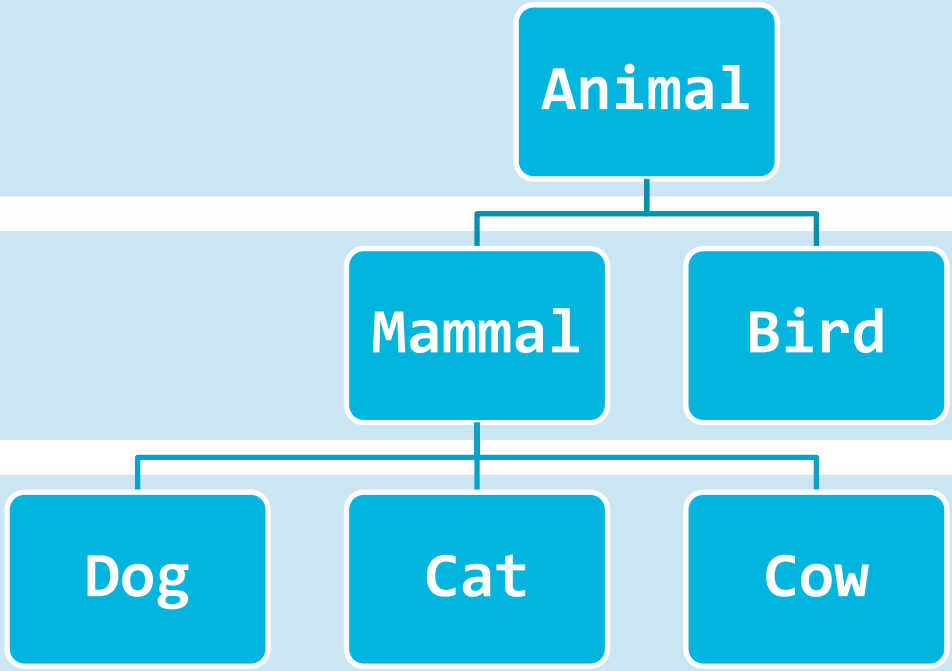


Herança – Hierarquia de Classes - vocabulário

`Animal` é **superclasse** direta (classe **base** ou classe **pai**) de **`Mammal`** e **`Bird`**, e é super classe indireta de **`Dog`**, **`Cat`** e **`Cow`**

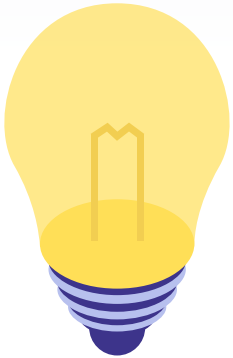
`Mammal` é **subclasse** direta (classe **derivada** ou classe **filha**) de **`Animal`** e super classe direta (base ou pai) de **`Dog`**, **`Cat`** e **`Cow`**

`Dog` é subclasse direta (classe derivada ou classe filha) de **`Mammal`** e indirecta de **`Animal`**



Herança

- As subclasses ou classes derivadas vão herdar todos os atributos e métodos da superclasse ou classe base.
- As subclasses não herdam os construtores da superclasse



Construtores em Herança

- Herança de Classes

Herança

- As subclasses **não herdam os construtores** da superclasse

```
public class Post {
    private String username;
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    public Post(String author) {
        username = author;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<String>();
    }

    public void like() {
        likes++;
    }
}

public class MessagePost extends Post {
    private String message;
}
```

```
MessagePost message2 = new MessagePost("Ana");
  Error: constructor MessagePost in class MessagePost cannot be applied to give
    required: java.lang.String,java.lang.String
    found: java.lang.String
    reason: actual and formal argument lists differ in length
```

```
MessagePost message = new MessagePost("Zé", "Olá");
message.like();
```

Herança

- As **subclasses** ou classes derivadas vão **herdar todos os atributos e métodos** da superclasse ou classe base.
- As subclasses **não herdam os construtores** da superclasse
- Neste caso é necessário lidar com os construtores de uma forma diferente:
  - Cada subclasse escolhe o construtor que vai usar da superclasse
  - A escolha é feita no construtor da subclasse utilizando-se o método `super()` que representa uma chamada ao construtor da `super` classe
  - A escolha do construtor é feita dependendo dos argumentos do método `super()` Neste caso o número e tipo dos argumentos deve corresponder ao número e tipo de argumentos do construtor que se pretende utilizar da superclasse
  - O método `super()` deve ser o primeiro método a ser chamado dentro do construtor da subclasse

Exemplo – Rede Social

```
public class Post {
    private String username;
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    public Post(String author) {
        username = author;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<String>();
    }

    // métodos omitidos
}
```

- **Construtores** em herança – classe **`Post`**

Exemplo – Rede Social

```
public class MessagePost extends Post {
    private String message;
    public MessagePost(String author, String text) {
        super(author);
        message = text;
    }

    // métodos omitidos
}
```

- **Construtores** em herança – classe **MessagePost**

Chamada ao construtor da superclasse (**Post**)

Construtores em herança

- Construtores em herança
  - O **construtor da subclasse** deve incluir sempre uma chamada ao construtor da superclasse
    - Se não for incluída o compilador coloca automaticamente uma chamada ao construtor sem argumentos da superclasse: **`super();`**
      - Apenas resulta se a superclasse tiver um construtor sem argumentos
  - A chamada ao construtor da superclasse **deve ser a primeira instrução** do construtor da subclasse.

```
public class MessagePost extends Post
{
    private String message;

    public MessagePost(String author, String text)
{
        super(author);
        message = text;
    }

    // métodos omitidos
}
```

A classe **`Post`** é a superclasse da classe **`MessagePost`**

Chamada ao construtor da classe **`Post`**

Primeira instrução do construtor

Herança

```
public class Post {
    private String username;
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    public Post(String author) {
        username = author;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<String>();
    }
}

public class MessagePost extends Post {
    private String message;

    public MessagePost(String author) {
        super(author);
        message = "";
    }

}
```

- As subclasses **não herdam os construtores** da superclasse

```
MessagePost message2 = new MessagePost("Ana");
```

Novo construtor

Herança por Extensão

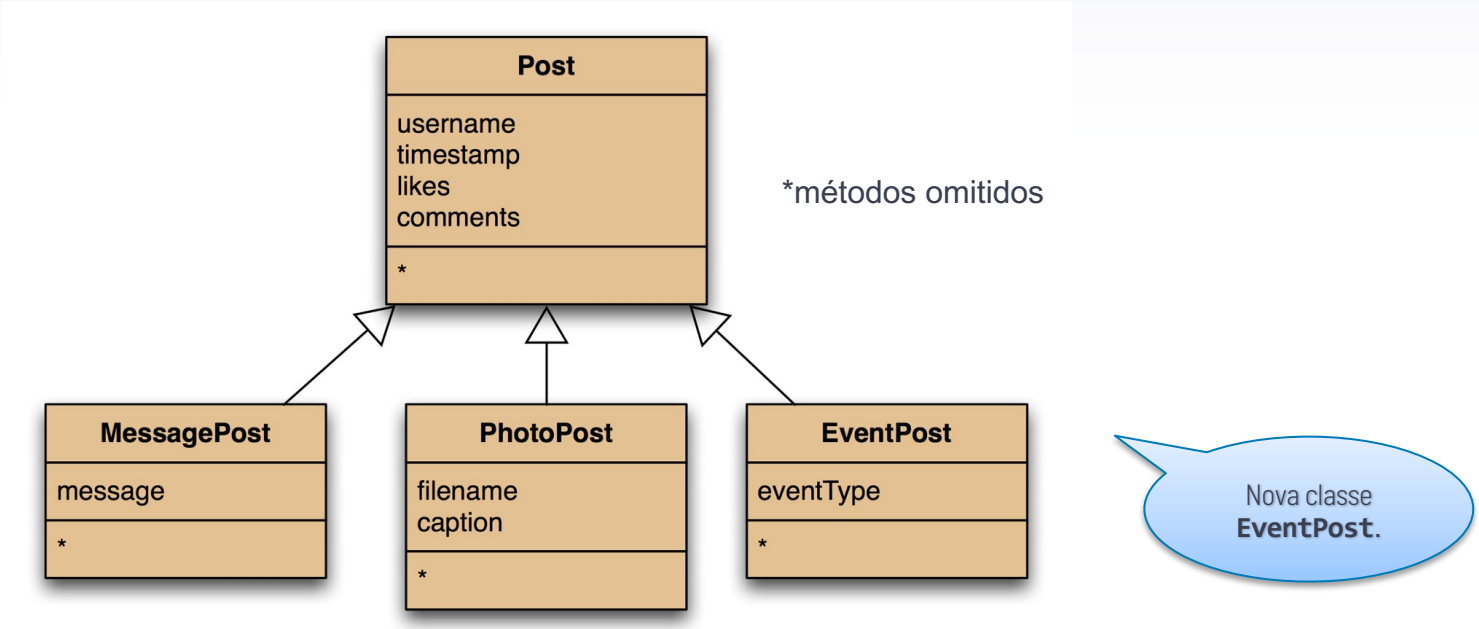
- Herança de Classes

Modificadores de acesso do Java

- Membros **public** (símbolo **+** nos diagramas de classe): Se os membros são declarados como públicos dentro de uma classe, então estes membros são acessíveis às classes que estão dentro e fora do pacote onde esta classe é visível. Este é o menos restritivo de todos os modificadores de acessibilidade.
- Membros **protected** (símbolo **#** nos diagramas de classe) : se os membros de uma classe são declarados como protegidos, eles estarão acessíveis a todas as classes do pacote e a todas as subclasses desta classe em qualquer pacote em que essa classe esteja visível.
- Membros **default** (nenhum símbolo) : quando nenhum modificador de acessibilidade é especificado para o membro, ele é implicitamente declarado como visibilidade por defeito. Eles são acessíveis apenas para as outras classes no pacote da classe.
- Membros **private** (símbolo **-** nos diagramas de classe) : Este é o mais restritivo de todos os modificadores de acessibilidade. Esses membros são acessíveis apenas dentro desta classe. Eles não são acessíveis a partir de nenhuma outra classe dentro do pacote da classe.

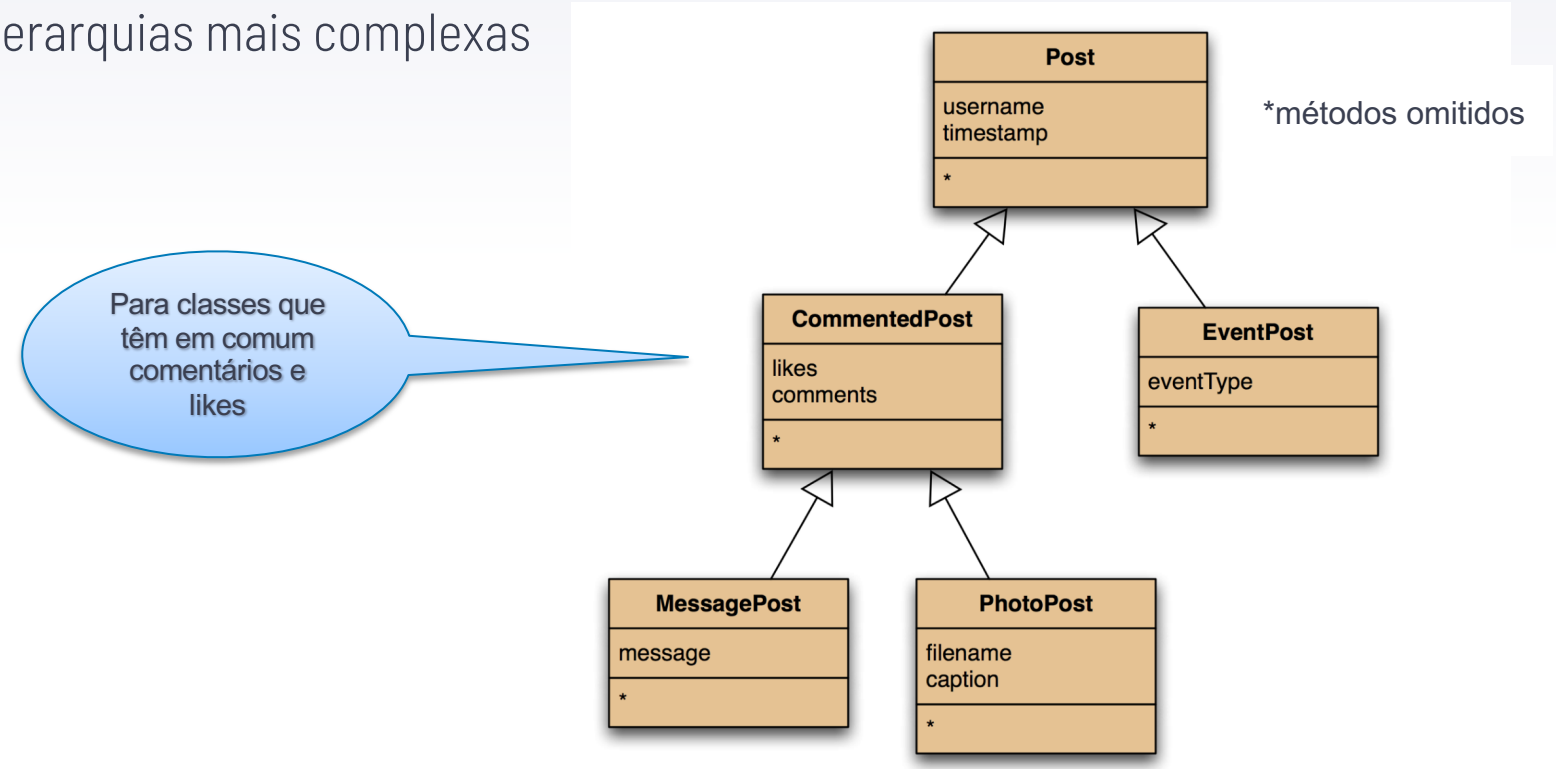
Exemplo – Rede Social

- Adição de outro tipo de Posts – **EventPost**



Exemplo – Rede Social

- Hierarquias mais complexas



Exemplo – Rede Social

Benefícios da **Herança de classes**

- Evita a duplicação de código
- Permite a reutilização de código
- Simplifica a manutenção
- Promove a extensão de classes

Principio da substituição

- Herança de Classes

Principio da Substituição

Subclasses e Subtipos

- Uma classe define um **tipo**
- Uma subclasse define um **subtipo**
- Sempre que é necessário um objeto de uma classe, pode-se usar em vez disso um objeto de uma subclasse:
  - Chama-se **principio da substituição**
    - Exemplo

```
Post post = new MessagePost("João", "Olá Mundo");
```

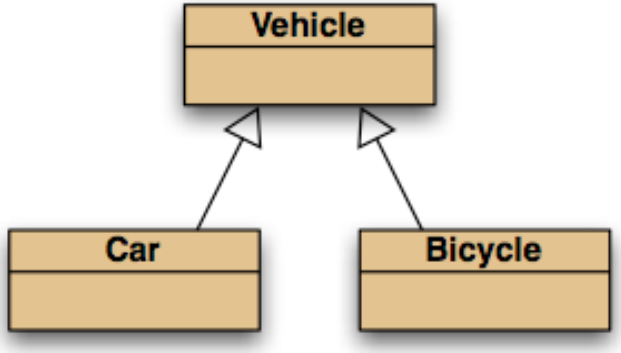
Guarda um objeto da classe **Post**

Atribui-se um objeto da subclasse **MessagePost**

Principio da Substituição

- Principio da substituição aplicado à atribuição de valores

```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```



Principio da Substituição

```
public class NewsFeed {

    public void addPost(Post post)
    {
        ...
    }
}

PhotoPost photo = new PhotoPost(...);
MessagePost message = new MessagePost(...);

feed.addPost(photo);
feed.addPost(message);
```

- Principio da substituição aplicado à passagem de parâmetros

Objetos de subclasses podem ser usados como parâmetros atuais para superclasses

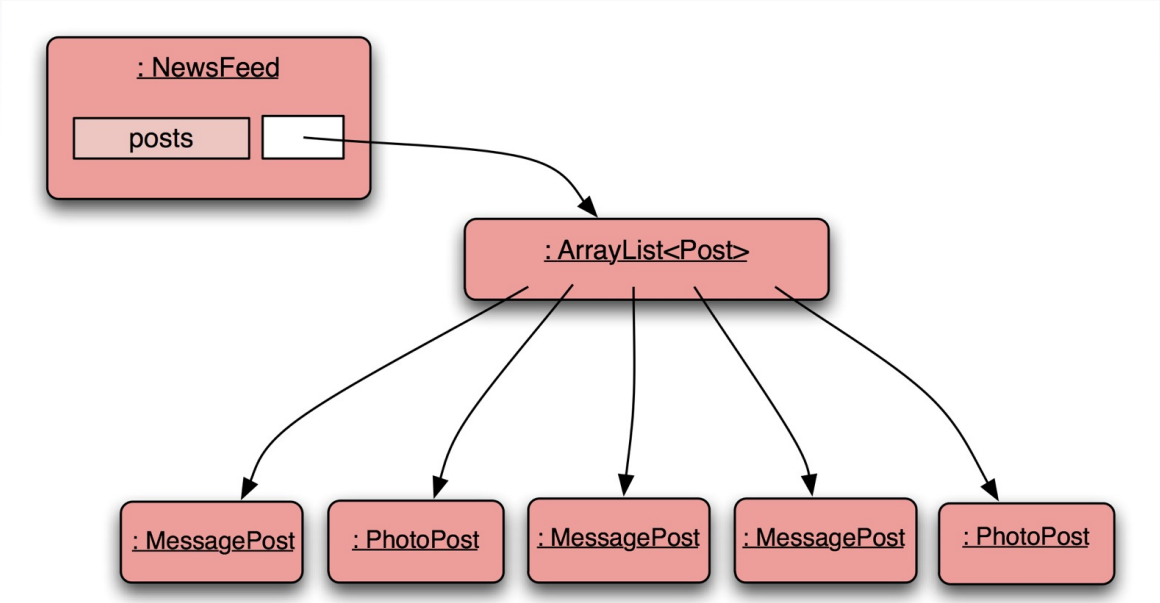
Exemplo – Rede Social

```
public class NewsFeed  {
    private ArrayList<Post> posts;

    public NewsFeed() {
        posts = new ArrayList<Post>();
    }

    public void addPost(Post post) {
        posts.add(post);
    }

    public void show() {
        for(Post post : posts) {
            post.display();
            System.out.println();
        }
    }
}
```

- Nova classe **NewsFeed**

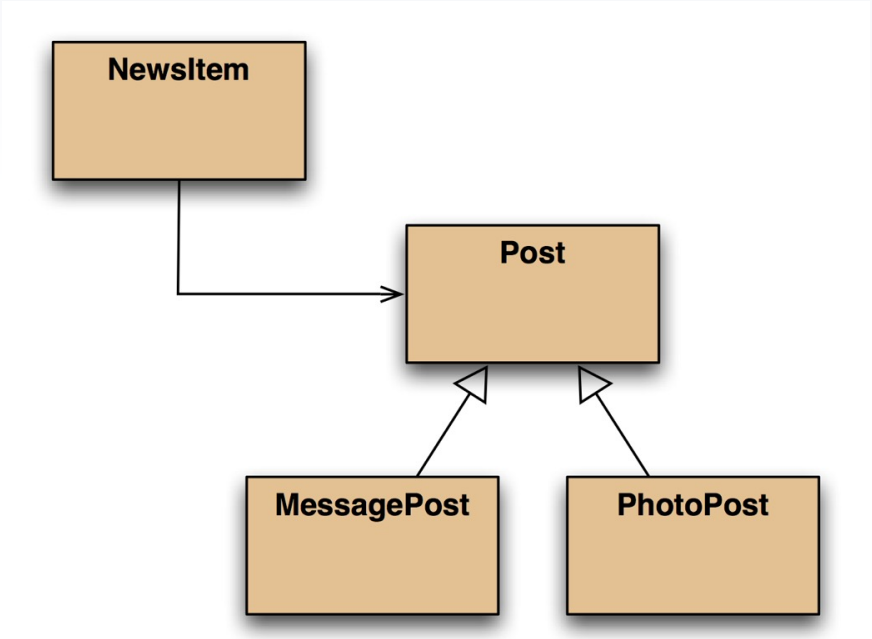
Exemplo – Rede Social

- Diagrama de objetos



Exemplo – Rede Social

- Diagrama de classes



Exemplo – Rede Social

- Antiga classe versus nova classe **NewsFeed**

```
public class NewsFeed {
    private ArrayList<MessagePost> messages;
    private ArrayList<PhotoPost> photos;
    public NewsFeed() {
        messages = new ArrayList<MessagePost>();
        photos = new ArrayList<PhotoPost>();
    }
    public void addMessagePost(MessagePost message)  {
        messages.add(message);
    }
    public void addPhotoPost(PhotoPost photo)  {
        photos.add(photo);
    }
    public void show()  {
        for(MessagePost message : messages) {
            message.display();
            System.out.println();
        }
        for(PhotoPost photo : photos) {
            photo.display();
            System.out.println();
        }
    }
}
```

```
public class NewsFeed  {
    private ArrayList<Post> posts;

    public NewsFeed() {
        posts = new ArrayList<Post>();
    }

    public void addPost(Post post) {
        posts.add(post);
    }

    public void show() {
        for(Post post : posts) {
            post.display();
            System.out.println();
        }
    }
}
```

Bibliografia

- Objects First with Java (6th Edition), David Barnes & Michael Kölling, Pearson Education Limited, 2016
  - Capítulo 10