

Programación Orientada a Objetos V - Herencia y Polimorfismo

Estructuras de Datos y Algoritmos /
Algoritmos y Estructuras de Datos II

Año 2025

Dr. Pablo Ponzio

Universidad Nacional de Río Cuarto
CONICET



Ejemplo: Network v1

- El proyecto network implementa un prototipo de una pequeña parte de una red social: el news feed, con la lista de mensajes que se muestran cuando se abre la aplicación
- Requisitos de la primera versión:
 - Dos tipos posibles de posts: mensajes de texto y fotos
 - Vamos a desarrollar el motor que almacena y muestra estos posts
 - Funcionalidades a proveer:
 - Creación de posts
 - Los posts de texto tienen un mensaje de longitud arbitraria
 - Los posts de fotos tienen una imagen (el path) y un pie de foto
 - Todos los posts tienen: usuario del autor, fecha de creación, número de likes, lista de comentarios
 - Búsqueda de posts: buscar los posts de un usuario, o de una fecha dada
 - Mostrar listas de posts (por terminal)
 - Eliminación de posts

Network v1: Diseño

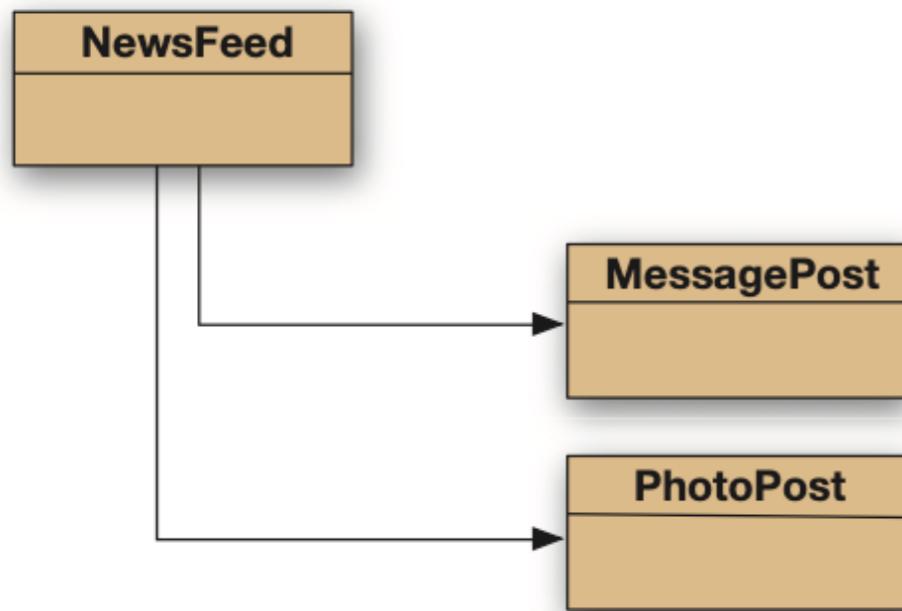


Diagrama de clases

Network v1: Diseño

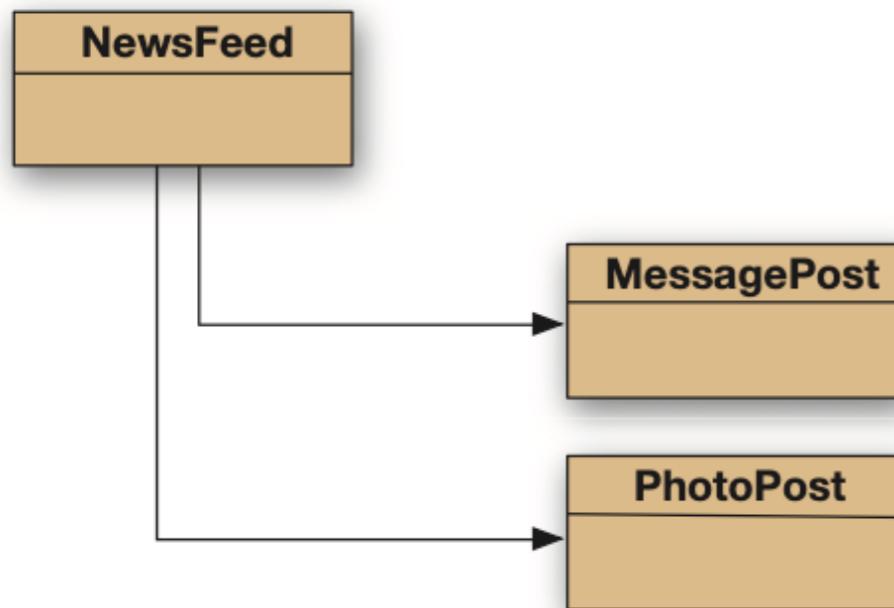
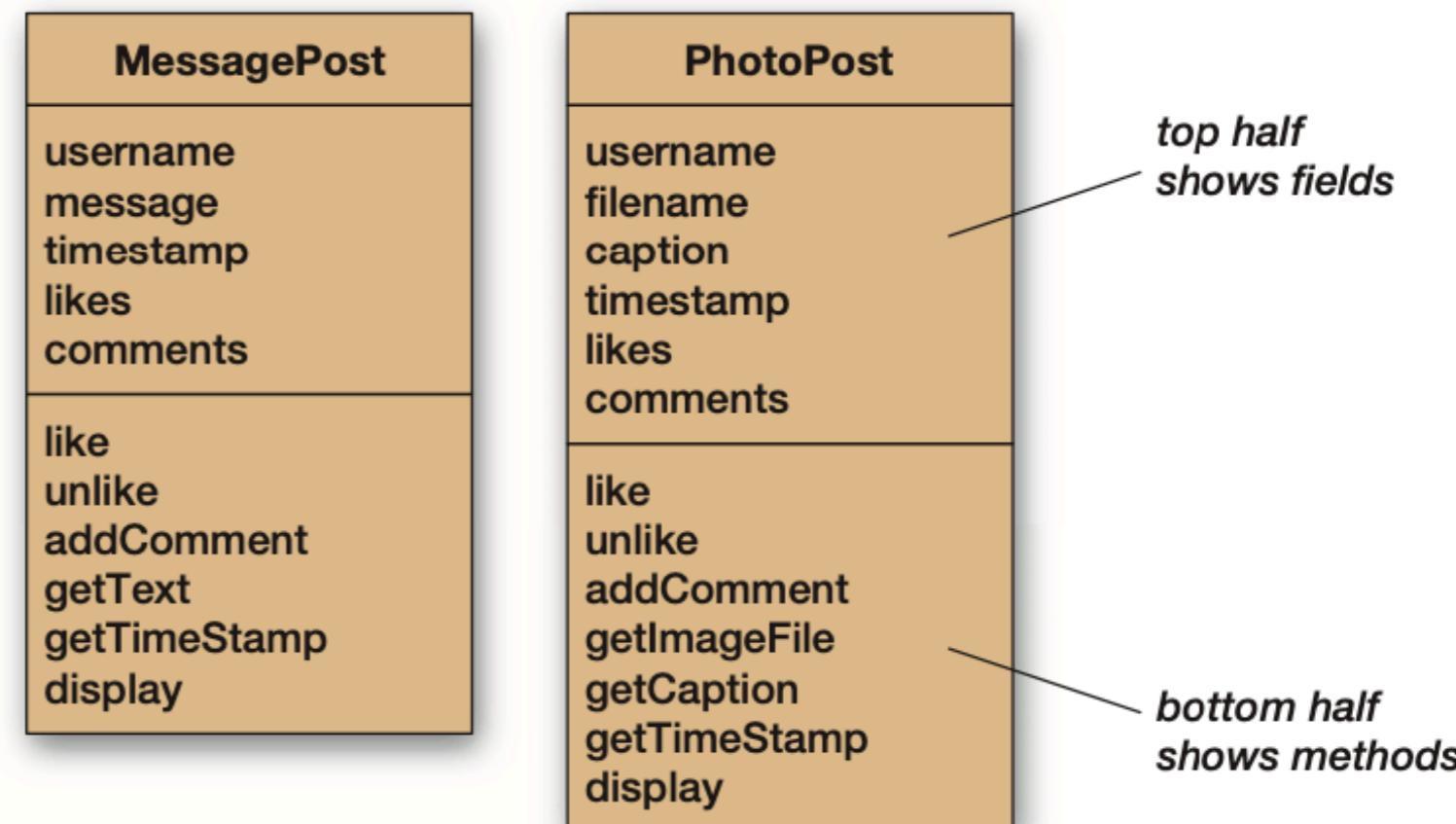
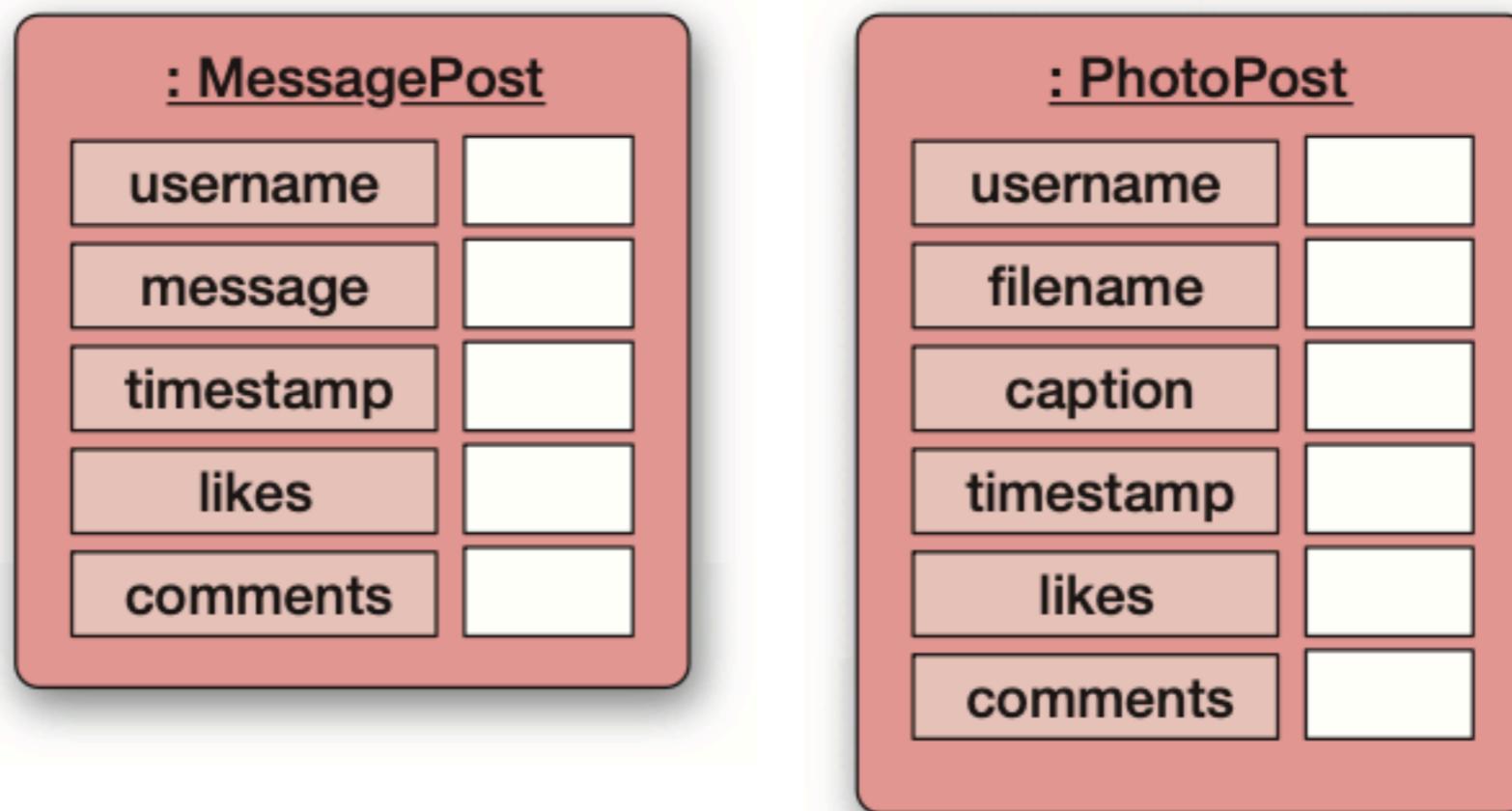


Diagrama de clases



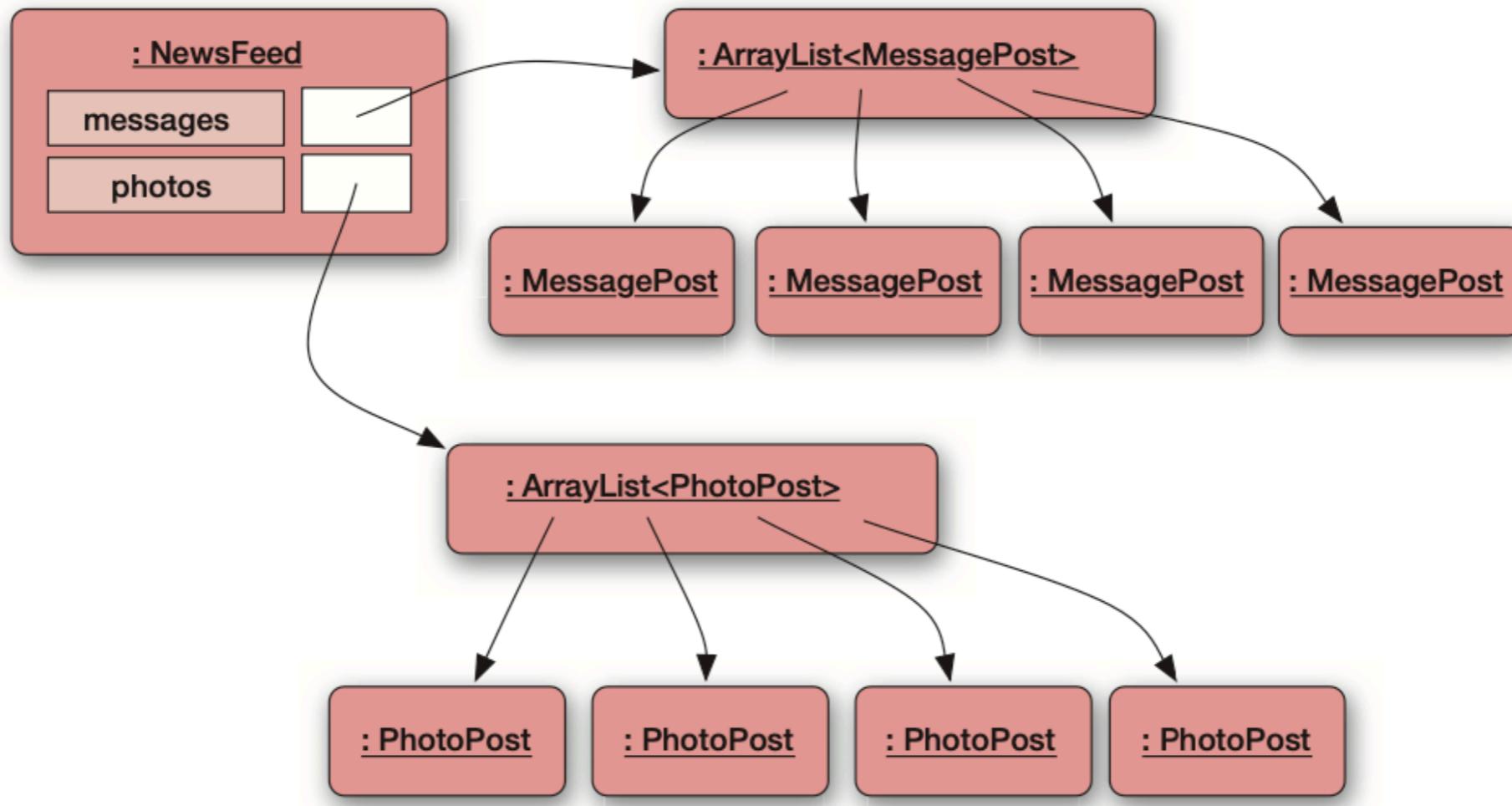
Vista detallada de algunas clases
(más usada en diseño OO)

Network v1: Diagramas de objetos



- Notar que los distintos tipos de post comparten mucha información

Network v1: Diagramas de objetos



- Usando los mecanismos vistos hasta el momento, necesitamos dos listas distintas para guardar los distintos tipos de posts

Network v1: MessagePost

```
/**  
 * This class stores information about a post in a social network.  
 * The main part of the post consists of a (possibly multi-line)  
 * text message. Other data, such as author and time, are also stored.  
 */  
public class MessagePost  
{  
    private String username; // username of the post's author  
    private String message; // an arbitrarily long, multi-line message  
    private long timestamp;  
    private int likes;  
    private ArrayList<String> comments;  
  
    /**  
     * @param author Constructs a MessagePost object with the given  
     * 'author' and 'text'.  
     */  
    public MessagePost(String author, String text)  
    {  
        username = author;  
        message = text;  
        timestamp = System.currentTimeMillis();  
        likes = 0;  
        comments = new ArrayList<>();  
    }  
}
```

Network v1: MessagePost

```
/**  
 * This class stores information about a post in a social network.  
 * The main part of the po/*  
 * text message. Other dat * @post Record one more 'Like' indication from a user.  
 */  
public class MessagePost {  
    private String username;  
    private String message;  
    private long timestamp;  
    private int likes;  
    private ArrayList<String> comments;  
  
    /**  
     * @post Constructs a new post with the specified author and text.  
     */  
    public MessagePost(String author, String text)  
    {  
        username = author;  
        message = text;  
        timestamp = System.currentTimeMillis();  
        likes = 0;  
        comments = new ArrayList<String>();  
    }  
  
    /**  
     * @post Record one more 'Like' indication from a user.  
     */  
    public void like()  
    {  
        likes++;  
    }  
  
    /**  
     * @post Record that a user has withdrawn his/her 'Like' vote.  
     */  
    public void unlike()  
    {  
        if (likes > 0) {  
            likes--;  
        }  
    }  
  
    /**  
     * @post Add a comment to this post.  
     */  
    public void addComment(String text)  
    {  
        comments.add(text);  
    }  
}
```

Network v1: MessagePost

```
/**  
 * This class stores information about a post in a social network.  
 * The main part of the post is the  
 * text message. Other data includes the author's name,  
 * the timestamp of creation, and the number of likes.  
 */  
public class MessagePost {  
  
    private String username;  
    private String message;  
    private long timestamp;  
    private int likes;  
    private ArrayList<String> comments;  
  
    /**  
     * @post Constructs a new MessagePost object.  
     * 'author' and 'text' are required parameters.  
     */  
    public MessagePost(String author, String text) {  
        username = author;  
        message = text;  
        timestamp = System.currentTimeMillis();  
        likes = 0;  
        comments = new ArrayList<String>();  
    }  
  
    /**  
     * @post Record one more 'Like' indication from a user.  
     */  
    public void like() {  
        likes++;  
    }  
  
    /**  
     * @post Record that a user has withdrawn his/her 'Like' vote.  
     */  
    public void unlike() {  
        if (likes > 0) {  
            likes--;  
        }  
    }  
  
    /**  
     * @post Add a comment to this post.  
     */  
    public void addComment(String text) {  
        comments.add(text);  
    }  
  
    /**  
     * @post Return the text of this post.  
     */  
    public String getText() {  
        return message;  
    }  
  
    /**  
     * @post Return the time of creation of this post.  
     */  
    public long getTimeStamp() {  
        return timestamp;  
    }  
}
```

Network v1: MessagePost

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
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.");  
    }  
}
```

Network v1: MessagePost

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public void display()  
{  
    System.out.println(username);  
    System.out.println(message);  
    System.out.print(timeString(timestamp));  
  
    if(likes > 0) {  
        System.out.println(" - " +  
    }  
    else {  
        System.out.println();  
    }  
  
    if(comments.isEmpty()) {  
        System.out.println(" No com");  
    }  
    else {  
        System.out.println(" " + co);  
    }  
}  
  
    /**  
     * @post Create a string describing a time point in the past in terms  
     * relative to current time, such as "30 seconds ago" or "7 minutes ago".  
     * Currently, only seconds and minutes are used for the string.  
     */  
    private String timeString(long time)  
{  
        long current = System.currentTimeMillis();  
        long pastMillis = current - time;          // time passed in milliseconds  
        long seconds = pastMillis/1000;  
        long minutes = seconds/60;  
        if(minutes > 0) {  
            return minutes + " minutes ago";  
        }  
        else {  
            return seconds + " seconds ago";  
        }  
    }
```

Network v1: PhotoPost

```
/**  
 * This class stores information about a post in a social network.  
 * The main part of the post consists of a photo and a caption.  
 * Other data, such as author and time, are also stored.  
 */  
public class PhotoPost  
{  
    private String username; // username of the post's author  
    private String filename; // the name of the image file  
    private String caption; // a one line image caption  
    private long timestamp;  
    private int likes;  
    private ArrayList<String> comments;  
  
    /**  
     * @post Constructs a PhotoPost object with the given 'author',  
     * the 'filename' of the image, and the 'caption'.  
     */  
    public PhotoPost(String author, String filename, String caption)  
    {  
        username = author;  
        this.filename = filename;  
        this.caption = caption;  
        timestamp = System.currentTimeMillis();  
        likes = 0;  
        comments = new ArrayList<>();  
    }
```

Network v1: PhotoPost

```
/**  
 * This class stores information about a post in a social network  
 * The main part of the    /**  
 * Other data, such as a   * @post Record one more 'Like' indication from a user.  
 */  
 */  
public class PhotoPost  
{  
    private String username;  
    private String filename;  
    private String caption;  
    private long timestamp;  
    private int likes;  
    private ArrayList<String> comments;  
  
    /**  
     * @post Constructs  
     * the 'filename' of  
     */  
    public PhotoPost(String filename)  
    {  
        username = autho;  
        this.filename = filename;  
        this.caption = c;  
        timestamp = Syst;  
        likes = 0;  
        comments = new A  
    }  
    /**  
     * @post Record one more 'Like' indication from a user.  
     */  
    public void like()  
    {  
        likes++;  
    }  
  
    /**  
     * @post Record that a user has withdrawn his/her 'Like' vote.  
     */  
    public void unlike()  
    {  
        if (likes > 0) {  
            likes--;  
        }  
    }  
  
    /**  
     * @post Add a comment to this post.  
     */  
    public void addComment(String text)  
    {  
        comments.add(text);  
    }  
}
```

Network v1: PhotoPost

```
/**  
 * This class stores information about a post in a social network  
 * The main part of the    /**  
 * Other data, such as a   * @post Record one more 'Like' indication from a user.  
 */  
public class PhotoPost  
{  
    private String username;  
    private String filename;  
    private String caption;  
    private long timestamp;  
    private int likes;  
    private ArrayList<String> comments;  
  
    /**  
     * @post Constructs  
     * the 'filename' of  
     */  
    public PhotoPost(String auth)  
    {  
        username = auth;  
        this.filename = "";  
        this.caption = "";  
        timestamp = System.currentTimeMillis();  
        likes = 0;  
        comments = new ArrayList<String>();  
    }  
  
    /**  
     * @post Record one more 'Like' indication from a user.  
     */  
    public void like()  
    {  
        likes++;  
    }  
  
    /**  
     * @post Record that a user has disliked this post.  
     */  
    public void unlike()  
    {  
        if (likes > 0) {  
            likes--;  
        }  
    }  
  
    /**  
     * @post Add a comment to this post.  
     */  
    public void addComment(String text)  
    {  
        comments.add(text);  
    }  
  
    /**  
     * @post Return the file name of the image in this post.  
     */  
    public String getImageFile()  
    {  
        return filename;  
    }  
  
    /**  
     * @post Return the caption of the image of this post.  
     */  
    public String getCaption()  
    {  
        return caption;  
    }  
  
    /**  
     * @post Return the time of creation of this post.  
     */  
    public long getTimeStamp()  
    {  
        return timestamp;  
    }  
}
```

Network v1: PhotoPost

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
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.");  
    }  
}
```

Network v1: PhotoPost

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
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(" - " + lik);  
    }  
    else {  
        System.out.println();  
    }  
  
    if(comments.isEmpty()) {  
        System.out.println(" No commen");  
    }  
    else {  
        System.out.println(" " + comme);  
    }  
}  
  
/**  
 * @post Create a string describing a time point in the past in terms  
 * relative to current time, such as "30 seconds ago" or "7 minutes ago".  
 * Currently, only seconds and minutes are used for the string.  
 */  
  
private String timeString(long time)  
{  
    long current = System.currentTimeMillis();  
    long pastMillis = current - time;          // time passed in milliseconds  
    long seconds = pastMillis/1000;  
    long minutes = seconds/60;  
    if(minutes > 0) {  
        return minutes + " minutes ago";  
    }  
    else {  
        return seconds + " seconds ago";  
    }  
}
```

Network v1: PhotoPost

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
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(" - " + lik  
    }  
    else {  
        System.out.println();  
    }  
  
    if(comments.isEmpty()) {  
        System.out.println(" No commen  
    }  
    else {  
        System.out.println(" " + comments.size() + " comments");  
    }  
}  
  
/**  
 * @post Create a string describing a time point in the past in terms  
 * relative to current time, such as "30 seconds ago" or "7 minutes ago".  
 * Currently, only seconds and minutes are used for the string.  
 */  
  
private String timeString(long time)  
{  
    long current = System.currentTimeMillis();  
    long pastMillis = current - time; // time passed in milliseconds  
    long seconds = pastMillis / 1000;  
    long minutes = seconds / 60;  
    long hours = minutes / 60;  
    long days = hours / 24;  
    long weeks = days / 7;  
    long months = weeks / 4;  
    long years = months / 12;  
  
    if(years > 0) {  
        return years + " years ago";  
    }  
    if(months > 0) {  
        return months + " months ago";  
    }  
    if(weeks > 0) {  
        return weeks + " weeks ago";  
    }  
    if(days > 0) {  
        return days + " days ago";  
    }  
    if(hours > 0) {  
        return hours + " hours ago";  
    }  
    if(minutes > 0) {  
        return minutes + " minutes ago";  
    }  
    if(seconds > 0) {  
        return seconds + " seconds ago";  
    }  
    return "just now";  
}
```

Hay mucho código duplicado en las clases MessagePost y PhotoPost: los atributos username, timestamp, likes y comments; los métodos like, unlike, addComment, getTimeStamp, timeString; y buena parte del código de display

Network v1: NewsFeed

```
/**  
 * The NewsFeed class stores news posts for the news feed in a social network  
 * application.  
 *  
 * Display of the posts is currently simulated by printing the details to the  
 * terminal. (Later, this should display in a browser.)  
 *  
 * This version does not save the data to disk, and it does not provide any  
 * search or ordering functions.  
 */  
public class NewsFeed  
{  
    private ArrayList<MessagePost> messages;  
    private ArrayList<PhotoPost> photos;  
  
    /**  
     * @post Construct an empty news feed.  
     */  
    public NewsFeed()  
    {  
        messages = new ArrayList<>();  
        photos = new ArrayList<>();  
    }  
}
```

Network v1: NewsFeed

```
/**  
 * The NewsFeed class stores news posts for the news  
 * application.  
 *  
 * Display of the posts is currently simulated by pri-  
 * terminal. (Later, this should display in a browser)  
 *  
 * This version does not save the data to disk, and i-  
 * search or ordering functions.  
 */  
public class NewsFeed  
{  
    private ArrayList<MessagePost> messages;  
    private ArrayList<PhotoPost> photos;  
  
    /**  
     * @post Construct an empty news feed.  
     */  
    public NewsFeed()  
    {  
        messages = new ArrayList<>();  
        photos = new ArrayList<>();  
    }
```

```
/**  
 * @post Add a text post to the news feed.  
 */  
public void addMessagePost(MessagePost message)  
{  
    messages.add(message);  
}  
  
/**  
 * @post Add a photo post to the news feed.  
 */  
public void addPhotoPost(PhotoPost photo)  
{  
    photos.add(photo);  
}
```

Network v1: NewsFeed

```
/**  
 * The NewsFeed class stores news posts for the news  
 * application.  
 *  
 * Display of the posts is currently simulated by pri  
 * terminal. (Later, this should display in a browser)  
 *  
 * This version does not save the data to disk, and i  
 * search or ordering function  
 */  
public class NewsFeed  
{  
    private ArrayList<MessagePost> messages;  
    private ArrayList<PhotoPost> photos;  
  
    /**  
     * @post Construct an empty news feed.  
     */  
    public NewsFeed()  
    {  
        messages = new ArrayList<MessagePost>();  
        photos = new ArrayList<PhotoPost>();  
    }  
  
    /**  
     * @post Add a text post to the news feed.  
     */  
    public void addMessagePost(MessagePost message)  
    {  
        messages.add(message);  
    }  
  
    /**  
     * @post Show the news feed. Currently: print the news feed details to the  
     * terminal. (To do: replace this later with display in web browser.)  
     */  
    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  
        }  
    }  
}
```

Network v1: NewsFeed

```
/**  
 * The NewsFeed class stores news posts for the news  
 * application.  
 *  
 * Display of the posts is currently simulated by pri  
 * terminal. (Later, this should display in a browser)  
 *  
 * This version does not save the data to disk, and i  
 * search or ordering function  
 */  
public class NewsFeed  
{  
    private ArrayList<MessagePost> messages;  
    private ArrayList<PhotoPost> photos;  
  
    /**  
     * @post Construct an empty news feed.  
     */  
    public NewsFeed()  
    {  
        messages = new ArrayList<MessagePost>();  
        photos = new ArrayList<PhotoPost>();  
    }  
  
    /**  
     * @post Add a text post to the news feed.  
     */  
    public void addMessagePost(MessagePost message)  
    {  
        messages.add(message);  
    }  
  
    /**  
     * @post Show the news feed. Currently: print the news feed details to the  
     * terminal. (To do: replace this later with display in web browser.)  
     */  
    public void show()  
    {  
        // display all text posts  
        for(MessagePost message : messages) {  
            message.display();  
            System.out.println(); // empty line between posts  
        }  
    }  
}
```

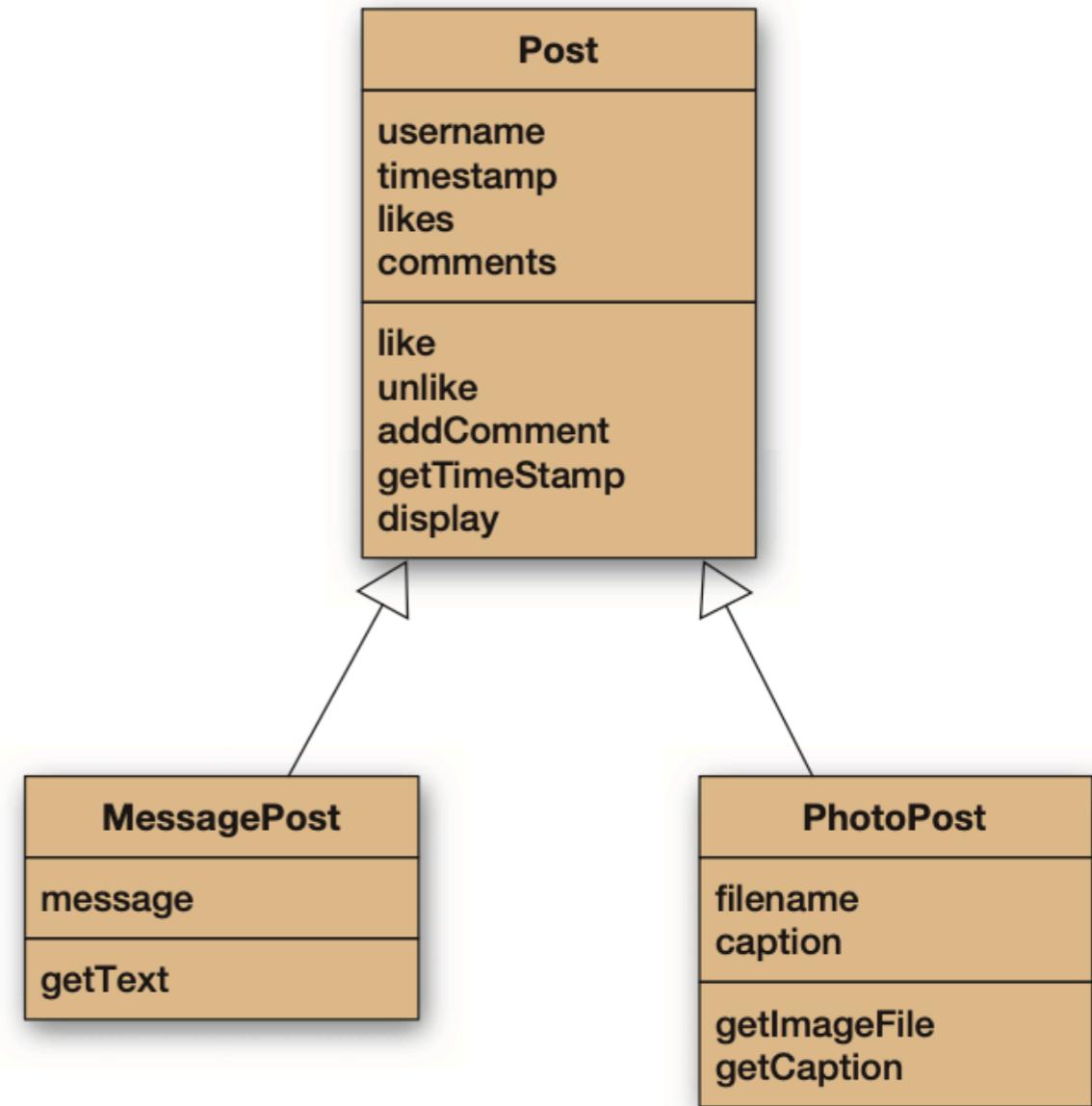
Hay código duplicado en NewsFeed: **dos ArrayLists, dos métodos add casi idénticos, dos recorridos de listas idénticos en show**

Evitar la duplicación de código

- El problema principal de nuestra implementación es el código duplicado
- La duplicación de código es problemática por varias razones:
 - Al escribir el mismo código varias veces podemos introducir errores involuntarios, que hagan que las distintas réplicas tengan distinto comportamiento
 - Dificulta significativamente la comprensibilidad del código y el mantenimiento:
 - Tenemos el doble de código (o más) para entender y modificar a la hora de corregir errores o agregar funcionalidades
 - Si hay mucho código duplicado es difícil saber exactamente qué clases tenemos que modificar
 - No hay que olvidarse de cambiar ninguno de los lugares donde el código está duplicado
 - Al agregar nuevas funcionalidades es posible que tengamos que seguir duplicando más y más el código (ej. al agregar un nuevo tipo de post)
- Uno de los principios de diseño de software más importantes es evitar la duplicación de código [1]
 - Hay distintos mecanismos para definir código reusable y evitar la replicación: abstracción procedural, abstracciones de datos, herencia (esta clase)

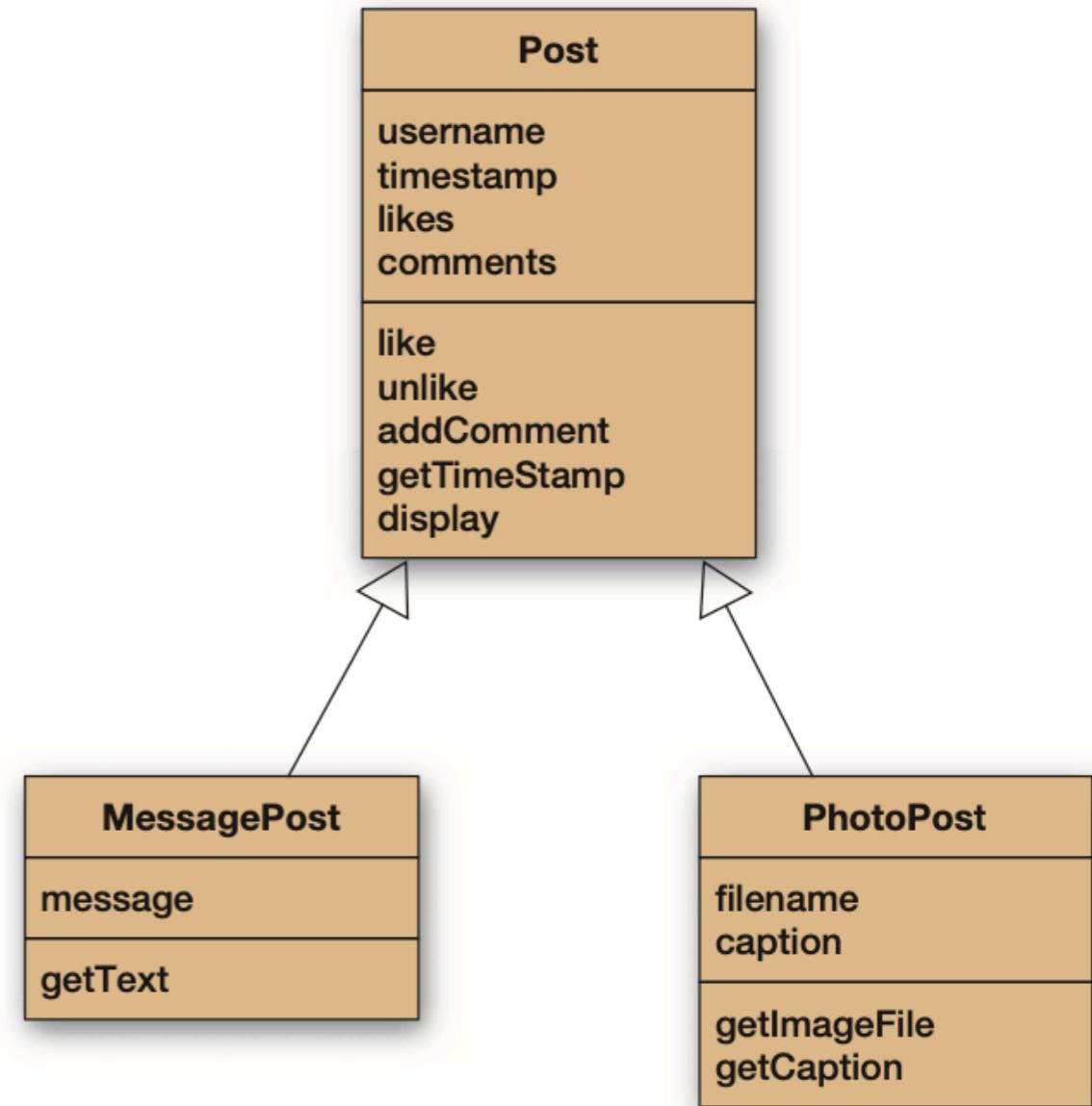
Herencia

- La herencia nos permite definir una clase como una extensión de otra
- En el diagrama de clases la relación de herencia se denota con una flecha con un triángulo sin relleno en la punta (ver Figura)
- Usando herencia podemos definir todos los atributos y métodos comunes en la superclase (Post)
- Y hacer que las subclases extiendan las superclases con sus propios atributos y métodos (MessagePost, PhotoPost, etc...)



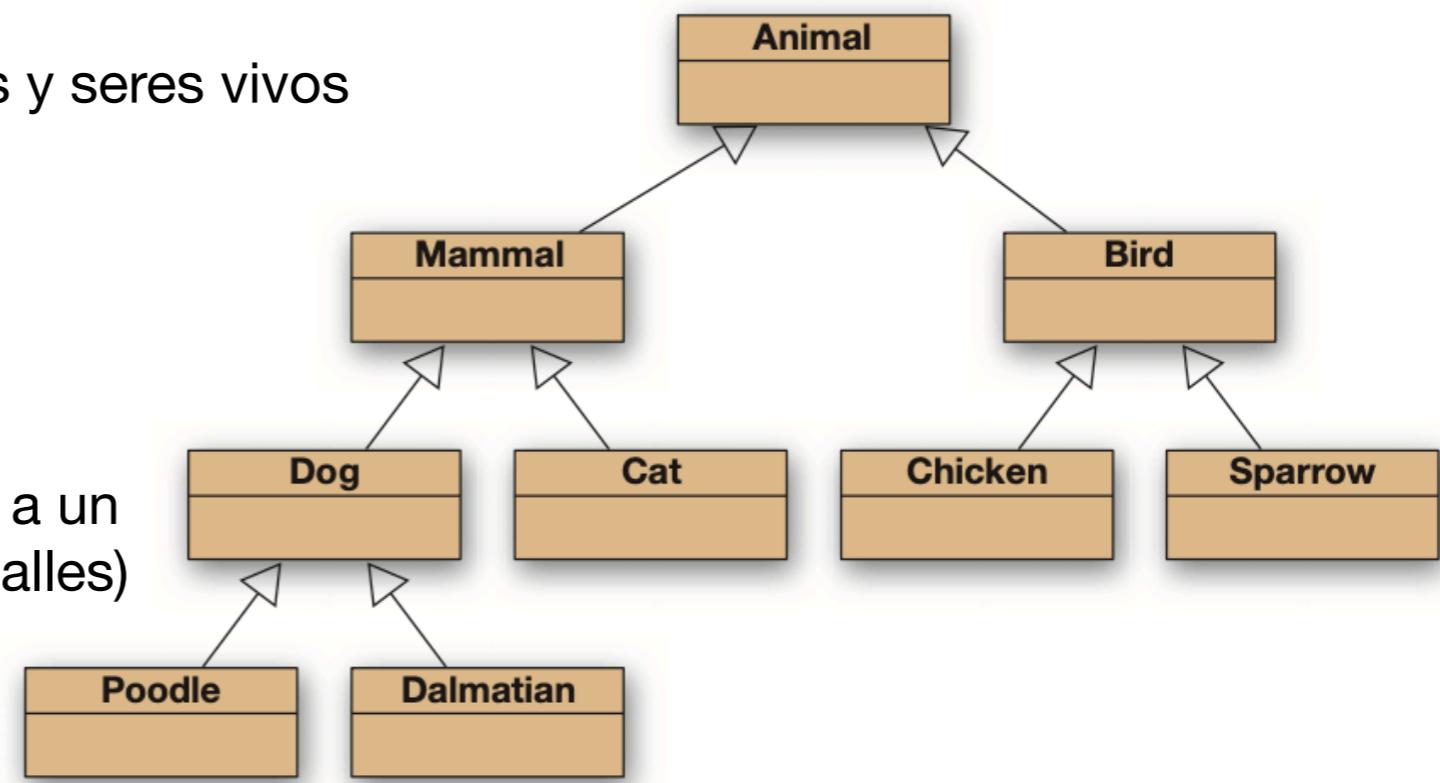
Herencia

- Las subclases tienen todos los atributos y todos los métodos de la superclase
 - Decimos que `MessagePost` y `PhotoPost` heredan el comportamiento de `Post`
- La herencia puede pensarse como una relación "es un"
 - `MessagePost` es un `Post`, `PhotoPost` es un `Post`



Jerarquías de herencia

- La herencia puede usarse para organizar clases en jerarquías complejas
- La jerarquía de herencia de la figura clasifica especies de animales
 - Por ejemplo, un caniche es un perro, y un mamífero y un animal
- La herencia es un mecanismo de abstracción que nos permite categorizar clases de objetos bajo ciertos criterios, y especificar las características de cada clase. Por ejemplo:
 - De los perros podemos saber que comen carne, que ladran, que son mamíferos y que son seres vivos
 - De los mamíferos que son mamíferos y seres vivos
 - De los animales que son seres vivos
- Si nos movemos hacia arriba en la jerarquía trabajamos a un nivel más abstracto (conocemos menos detalles)
- Si nos movemos hacia abajo nos vamos a un nivel más concreto (conocemos más detalles)
 - Es decir, las subclases son especializaciones de las superclases



Herencia en Java

- Para definir una clase que hereda de otra en Java usamos la palabra clave extends

```
public class Post
{
    private String username; // username of the post's author
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    // Constructors and methods omitted.
}

public class MessagePost extends Post
{
    private String message;
    // Constructors and methods omitted.
}

public class PhotoPost extends Post
{
    private String filename;
    private String caption;
    // Constructors and methods omitted.
}
```

- La superclase define los atributos y métodos comunes, y cada subclase define sus propios atributos

Herencia y modificadores de acceso

- Los objetos de la subclases tienen todos los atributos y métodos de las superclases
- Sin embargo:
 - Los atributos privados (`private`) de las superclases no pueden accederse en el código de las subclases
 - Los métodos privados (`private`) de las superclases no pueden invocarse en el código de las subclases
- El modificador de acceso `protected` permite definir atributos y métodos que pueden usarse en la clase que los define, en las subclases, y dentro de las clases del mismo paquete
 - Ejemplo: Si definimos en `Post`:

```
protected int likes;
```

`MessagePost` y `PhotoPost` pueden acceder y modificar el atributo `likes`
- Notar que hasta el momento no hemos definido paquetes, y por lo tanto estamos trabajando en el paquete por defecto
 - Todas las clases del paquete por defecto pueden acceder a los atributos `protected`
 - Esto implica que el ocultamiento de información definido en Java no es óptimo en muchos casos
 - Para evitar problemas, no debemos acceder directamente a los atributos de las clases (sino usar sus métodos)

Herencia y creación de objetos

- Cuando se crean objetos de una subclase, deben inicializarse los atributos de la superclase
 - Usualmente, estos atributos se inicializan usando el constructor de la superclase
- La clase `Post` se define para guardar los atributos y las funcionalidades comunes a todos los posts
- En esta aplicación, no tiene sentido crear objetos de la clase `Post`
- Vamos a definir el constructor de `Post` como `protected` para restringir su acceso a las subclases
 - Y al paquete que define `Post`, aunque nos gustaría ser más restrictivos
- Además, vamos a indicar en la pre del constructor que debería ser invocado sólo por sus subclases

```
public class Post
{
    private String username; // username of the post's author
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    /**
     * @pre Should be only called by Post's subclasses.
     * @post Constructor for objects of class Post.
     */
    protected Post(String author)
    {
        username = author;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<>();
    }
}
```

Herencia y creación de objetos

- En Java, los constructores de las subclases deben invocar siempre al constructor de la superclase en su primera línea
 - La palabra reservada `super` hace referencia al constructor de la superclase

```
public class MessagePost extends Post
{
    private String message; // an arbitrarily long, multi-line message

    /**
     * @post Constructs a MessagePost object with the given
     * 'author' and 'text'.
     */
    public MessagePost(String author, String text)
    {
        super(author);
        message = text;
    }
}
```

- Si el código no incluye la llamada a `super`, el compilador de Java intentará incluirla automáticamente
 - Y reportará un error en caso de no poder hacerlo (ej. el constructor tiene parámetros)

Herencia y creación de objetos

- En Java, los constructores de las subclases deben invocar siempre al constructor de la superclase en su primera línea
 - La palabra reservada `super` hace referencia al constructor de la superclase

```
public class MessagePost extends Post
{
    private String message; // an optional message

    /**
     * @post Constructs a MessagePost object with the given 'author',
     * 'author' and 'text'.
     */
    public MessagePost(String author, String text)
    {
        super(author);
        message = text;
    }
}

public class PhotoPost extends Post
{
    private String filename; // the name of the image file
    private String caption; // a one line image caption

    /**
     * @post Constructs a PhotoPost object with the given 'author',
     * the 'filename' of the image, and the 'caption'.
     */
    public PhotoPost(String author, String filename, String caption)
    {
        super(author);
        this.filename = filename;
        this.caption = caption;
    }
}
```

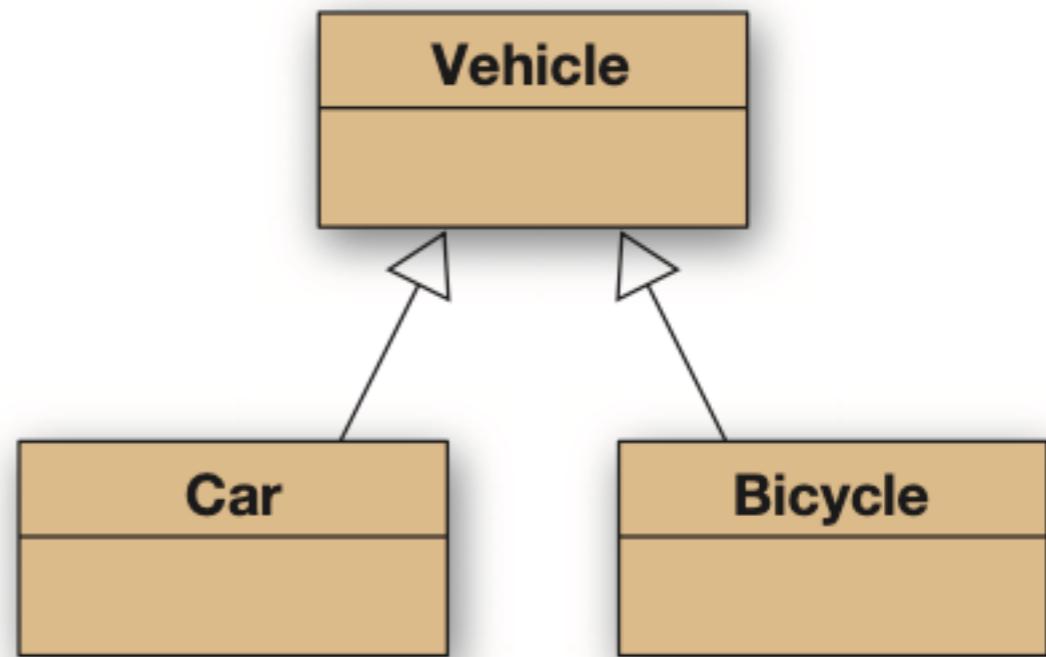
- Si el código no incluye la llamada a `super`, el compilador de Java intentará incluirla automáticamente
 - Y reportará un error en caso de no poder hacerlo (ej. el constructor tiene parámetros)

Subtipos y sustitución

- Notar que cuando usamos herencia estamos definiendo también jerarquías de tipos
- Decimos que el tipo definido por una subclase es un subtipo del tipo de la superclase
 - Por ejemplo, en la Figura Car y Bicycle son subtipos del tipo Vehicle
- Cuando se espera un objeto de un supertipo podemos sustituirlo con un objeto de cualquier subtipo
 - Esto se conoce como principio de sustitución
- Por ejemplo, las siguientes asignaciones son válidas por el principio de sustitución:

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

- Este principio se aplica a cualquier variable: atributos, parámetros, variables locales
- El tipo de la variable determina los métodos a los que podemos acceder
 - Sólo a los métodos de Vehicle para v1, v2 y v3



Polimorfismo

- Las variables de tipo objeto se denominan variables polimórficas, porque pueden hacer referencia a objetos de distintos tipos
 - De cualquier subtipo del tipo declarado
- Esto se denomina polimorfismo: las variables pueden tomar "distintas formas"
- La figura muestra la definición de la clase NewsFeed usando polimorfismo
- Usamos una única ArrayList para guardar los diferentes tipos de Posts
- Y un único método addPost, que toma como parámetro un objeto de la superclase Post, y lo agrega a la lista
- Un ejemplo de uso de este código se muestra a continuación:

```
NewsFeed feed = new NewsFeed();
MessagePost message = new MessagePost(. . .);
PhotoPost photo = new PhotoPost(. . .);
feed.addPost(message);
feed.addPost(photo);
```

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

    /**
     * @post Construct an empty news feed.
     */
    public NewsFeed()
    {
        posts = new ArrayList<>();
    }

    /**
     * @post Add a post to the news feed.
     */
    public void addPost(Post post)
    {
        posts.add(post);
    }
}
```

Polimorfismo

- Podemos definir también el método `show` de `NewsFeed` usando polimorfismo:

```
/**  
 * @post Show the news feed. Currently: print the news feed details  
 * to the terminal. (To do: replace this later with display  
 * in web browser.)  
 */  
public void show()  
{  
    // display all posts  
    for(Post post : posts) {  
        post.display();  
        System.out.println(); // empty line between posts  
    }  
}
```

- Este método tiene un problema: Invoca al método `display` de la clase `Post`, que sólo imprime los detalles comunes a todos los posts
 - Pero desconoce los detalles específicos de cada tipo de `Post`, y los omite
 - Vamos a ver como resolver esto más adelante en esta clase

Polimorfismo

- Podemos definir también el método `show` de `NewsFeed` usando polimorfismo:

```
public void display()
{
    System.out.println(username);
    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.");
    }
}
```

- Este método tiene un problema: Invoca al método `display` de la clase `Post`, que sólo imprime los detalles comunes a todos los posts
 - Pero desconoce los detalles específicos de cada tipo de `Post`, y los omite
 - Vamos a ver como resolver esto más adelante en esta clase

Ventajas de herencia

- Hasta el momento vimos que la herencia nos permite reusar código, extendiendo de una superclase para heredar su funcionalidad
 - Nos ayuda a eliminar la duplicación
- Otra de las ventajas de herencia, es que nos permite diseñar aplicaciones fáciles de extender
- Por ejemplo, la clase NewsFeed funciona correctamente con cualquier tipo de post
 - Siempre que satisfaga la especificación de Post
- De esta manera, si queremos agregar un nuevo tipo de post a nuestra aplicación, podemos hacerlo agregando nuevas clases, sin modificar el código existente

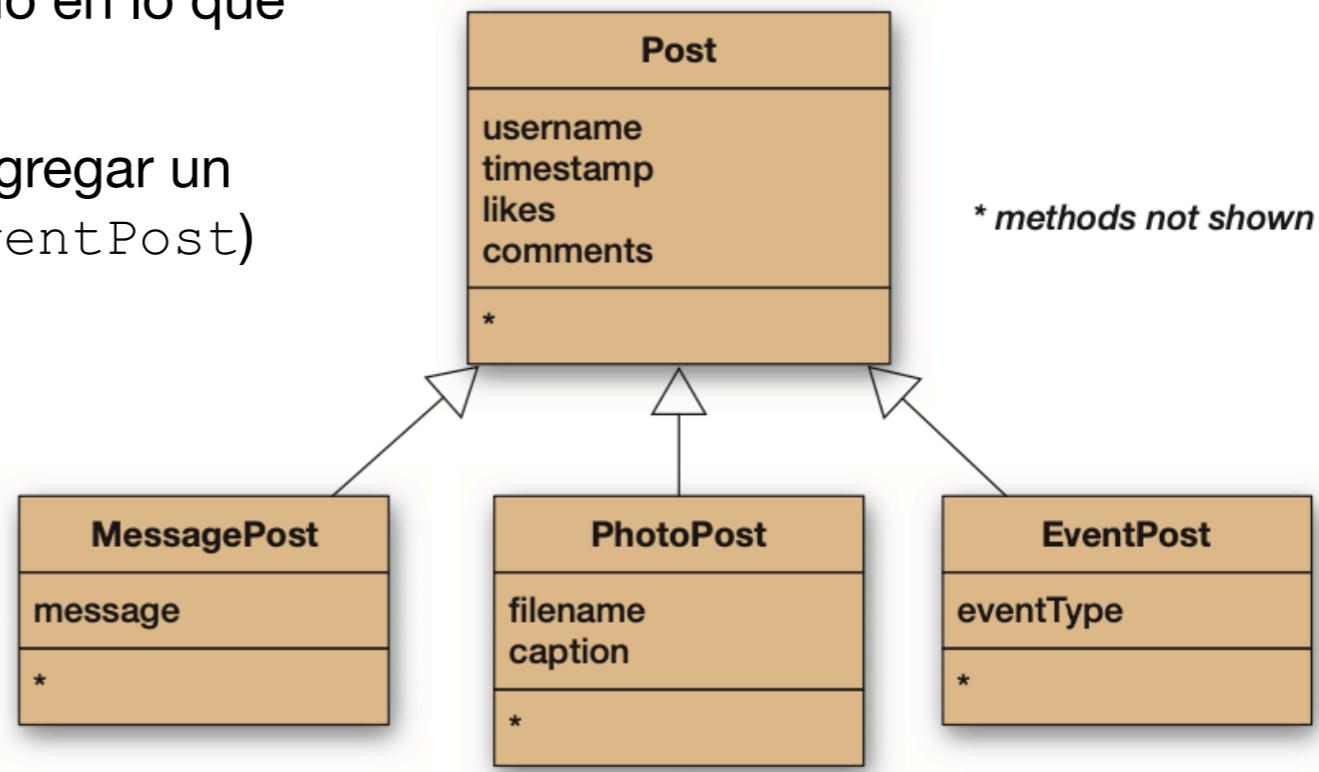
```
public class NewsFeed
{
    private ArrayList<Post> posts;

    /**
     * @post Construct an empty news feed.
     */
    public NewsFeed()
    {
        posts = new ArrayList<>();
    }

    /**
     * @post Add a post to the news feed.
     */
    public void addPost(Post post)
    {
        posts.add(post);
    }
}
```

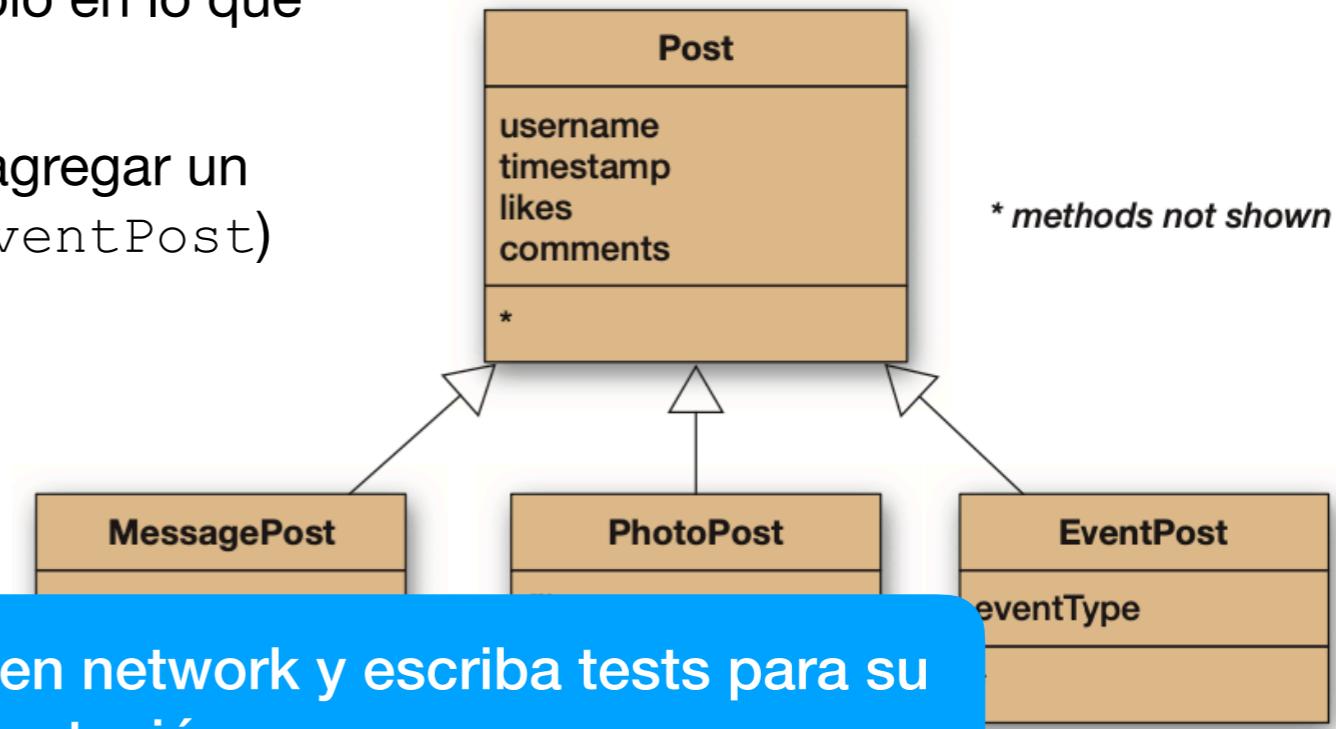
Principio abierto-cerrado

- El principio abierto-cerrado [1] (*open-closed principle*) dice que el código debe estar abierto para extensión, pero cerrado para modificación
 - Se extiende el código agregando nuevas clases
 - No se permite hacer cambios al código existente
- Es uno de los principios de diseño más importantes [1], y debemos intentar seguirlo tanto como sea posible para desarrollar aplicaciones fáciles de extender
- El diseño de network adhiere a este principio en lo que respecta a los tipos de posts soportados
- Por ejemplo, supongamos que queremos agregar un nuevo tipo de post que modele eventos (`EventPost`)
- Para hacerlo simplemente tenemos que agregar una nueva clase `EventPost`, que implemente este tipo de posts
 - Y dejar el resto del código sin modificaciones



Principio abierto-cerrado

- El principio abierto-cerrado [1] (*open-closed principle*) dice que el código debe estar abierto para extensión, pero cerrado para modificación
 - Se extiende el código agregando nuevas clases
 - No se permite hacer cambios al código existente
- Es uno de los principios de diseño más importantes [1], y debemos intentar seguirlo tanto como sea posible para desarrollar aplicaciones fáciles de extender
- El diseño de network adhiere a este principio en lo que respecta a los tipos de posts soportados
- Por ejemplo, supongamos que queremos agregar un nuevo tipo de post que modele eventos (`EventPost`)
- Para hacerlo simplemente tenemos que agregar una nueva clase `EventPost`, que implemente este tipo de posts
 - Y



Ejercicio: Implemente `EventPost` en `network` y escriba tests para su implementación

Problema con el método show

- Vamos a crear un test para `show` de `NewsFeed`, con los mensajes a continuación:

[Leonardo da Vinci](#)

Had a great idea this morning.

But now I forgot what it was. Something to do with flying . . .

40 seconds ago. 2 people like this.

No comments.

[Alexander Graham Bell](#)

[experiment.jpg] I think I might call this thing 'telephone'.

12 minutes ago. 4 people like this.

No comments.

Problema con el método show

- Vamos a crear un test para `show` de `NewsFeed`, con los mensajes a continuación:

Leonardo da Vinci

Had a great idea this morning.

But now I forgot what it was. Something

40 seconds ago. 2 people like this.

No comments.

Alexander Graham Bell

[experiment.jpg] I think I might

12 minutes ago. 4 people like this

No comments.

```
@Test
public void testShow() {
    MessagePost messagePost = new MessagePost("Leonardo da Vinci",
        "Had a great idea this morning.\n" +
        "But now I forgot what it was. Something to do with flying...");  

    messagePost.like();
    messagePost.like();
    PhotoPost photoPost = new PhotoPost("Alexander Graham Bell",
        "experiment.jpg",
        "I think I might call this thing 'telephone'.");
    photoPost.like();
    photoPost.like();
    photoPost.like();
    photoPost.like();
    NewsFeed feed = new NewsFeed();
    feed.addPost(messagePost);
    feed.addPost(photoPost);
    feed.show();
}
```

Problema con el método show

- El resultado de ejecutar el test es el siguiente:

Leonardo da Vinci
0 seconds ago - 2 people like this.
No comments.

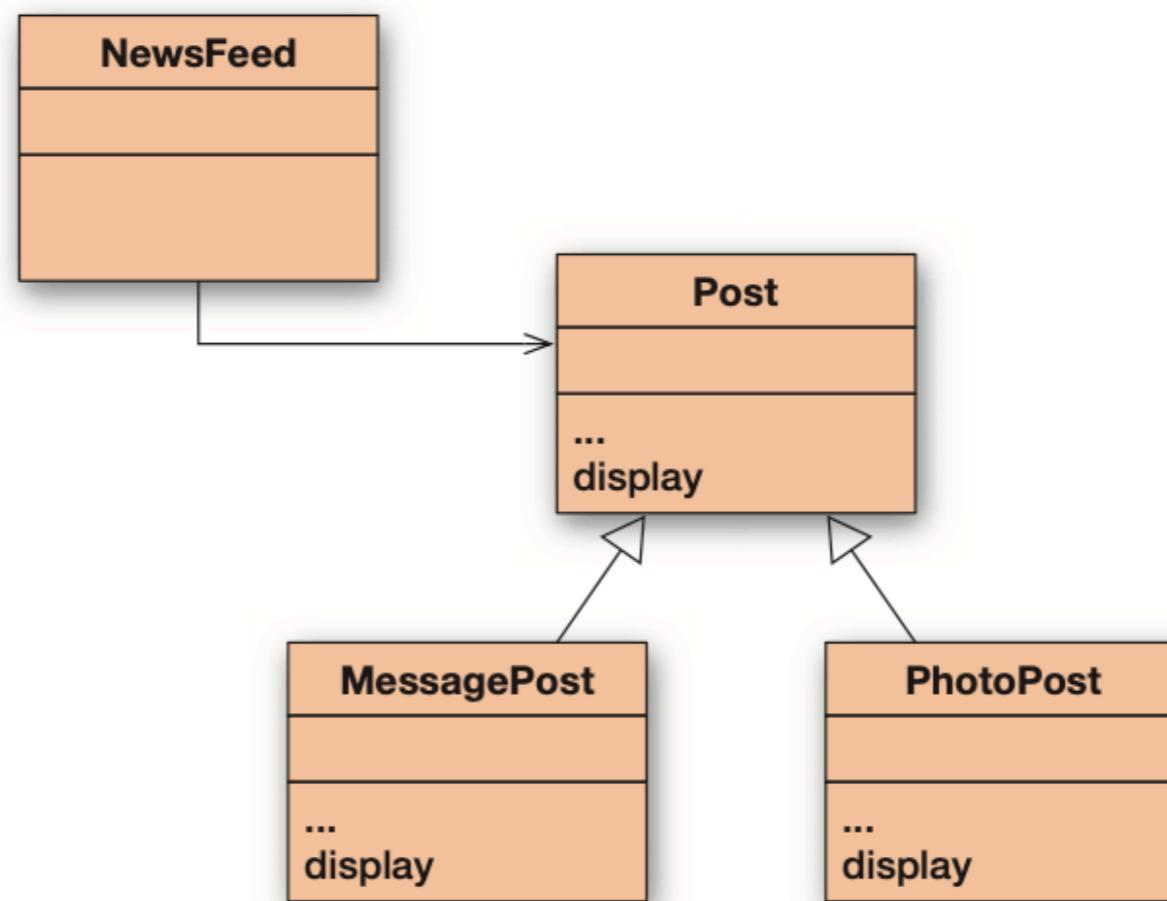
Alexander Graham Bell
0 seconds ago - 4 people like this.
No comments.

- Como `show` invoca al método `display` de la clase `Post`, los detalles particulares de cada tipo de mensaje no se imprimen
 - Estos detalles son desconocidos para la clase `Post`
 - Vamos a crear la versión 3 de network para solucionar este problema

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

Redefinición de métodos

- La solución consiste en redefinir el método `display` en las subclases de `Post`



- En inglés esto se llama `method overriding`

Network v3

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public void display()  
{  
    System.out.println(message);  
    super.display();  
}
```

En la clase MessagePost

- Redefinir un método consiste en definir un método con el mismo nombre en la subclase
- Podemos agregar la anotación `@Override` arriba del perfil del método para hacer explícita la redifinición en el código de la subclase

Network v3

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public void display()  
{  
    System.out.println(mes  
    super.display();  
}
```

En la clase MessagePost

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public void display()  
{  
    System.out.println(filename);  
    System.out.println(caption);  
    super.display();  
}
```

En la clase PhotoPost

- Redefinir un método consiste en definir un método con el mismo nombre en la subclase
- Podemos agregar la anotación `@Override` arriba del perfil del método para hacer explícita la redifinición en el código de la subclase

Network v3

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public void display()  
{  
    System.out.println(mes  
    super.display();  
}  
  
En la clase MessagePost  
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public void display()  
{  
    System.out.println(filename);  
    System.out.println(caption);  
    super.display();  
}
```

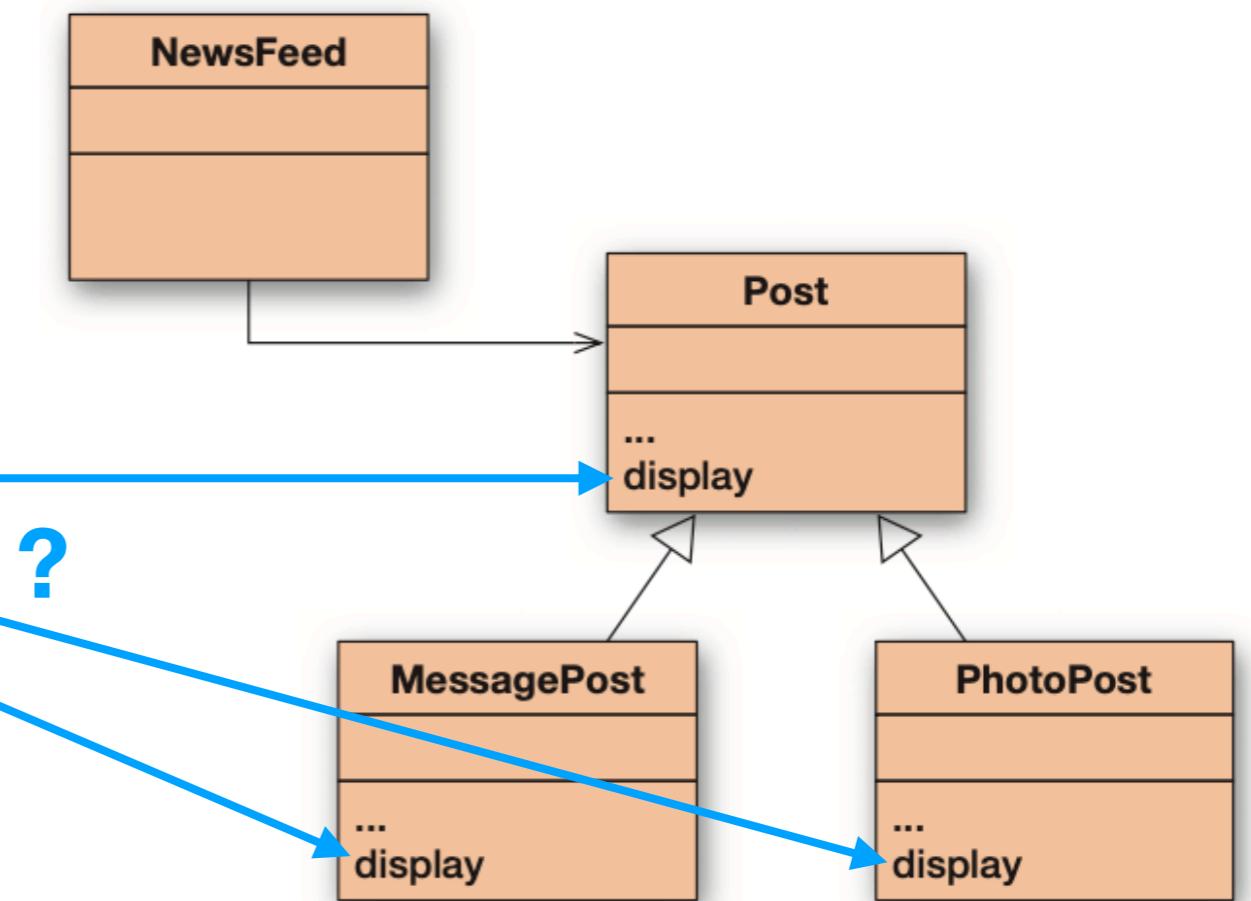
En la clase MessagePost
super puede usarse
para invocar al método redefinido de la
superclase

- Redefinir un método consiste en:
- Podemos agregar la anotación `@Override` arriba del perfil del método para hacer explícita la redifinición en el código de la subclase

Redefinición de métodos

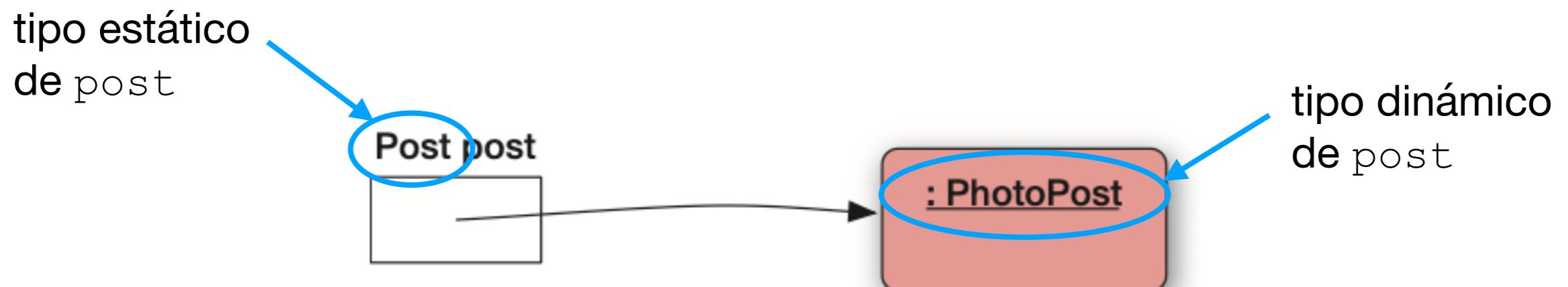
- ¿Cómo sabe Java a qué método invocar cuando hay redefinición de métodos?

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



Tipo estático y tipo dinámico de una variable

- El tipo (estático) de una variable es el tipo declarado de la variable en el código fuente
- El tipo dinámico de una variable es el tipo del objeto almacenado en la variable en un punto de la ejecución del programa

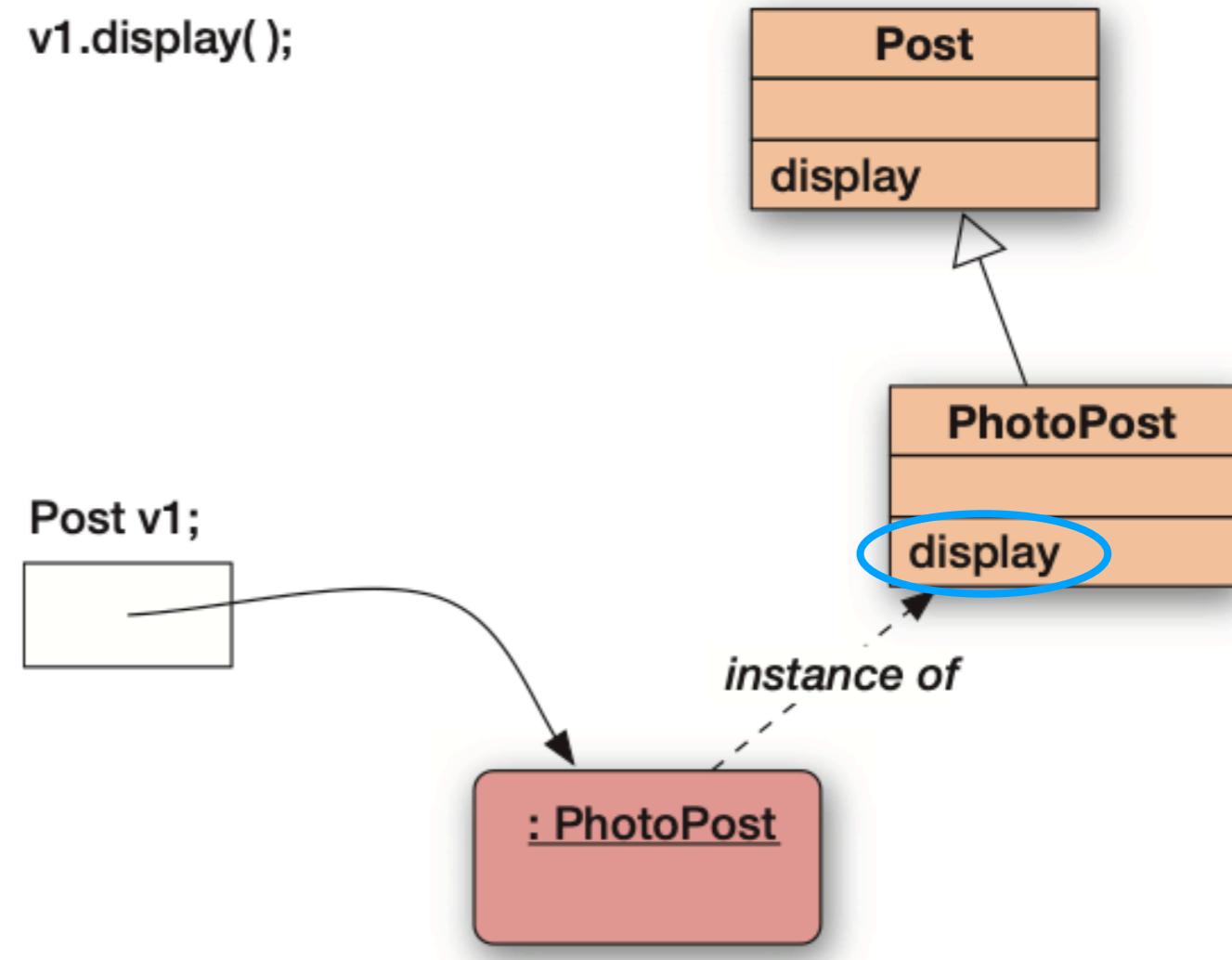


Dynamic dispatch

- El chequeo de tipos que realiza el compilador se basa en los tipos declarados de las variables en el código fuente (tipos estáticos)
- En cambio, en tiempo de ejecución del programa, la selección de qué método ejecutar depende del tipo dinámico de una variable
 - Se busca la clase del objeto referenciado por la variable, y se ejecuta el método correspondiente de la clase
 - Esto se denomina dynamic dispatch
- En la Figura, cuando se ejecuta `v1.display()` se busca la clase del objeto asociado a `v1` en tiempo de ejecución (`PhotoPost`), y se invoca a `display()` de `PhotoPost`
- Debido a esto, las invocaciones a métodos en Java son polimórficas: la misma invocación en distintos momentos de la ejecución puede ejecutar métodos diferentes
 - Esto se llama polimorfismo de métodos

`v1.display();`

`Post v1;`



Network v3: dynamic dispatch

```
@Test
public void testShow() {
    MessagePost messagePost = new MessagePost("Leonardo da Vinci",
        "Had a great idea this morning.\n" +
        "But now I forgot what it was. Something to do with flying...");  

    messagePost.like();
    messagePost.like();
    PhotoPost photoPost = new PhotoPost("Alexander Graham Bell",
        "experiment.jpg",
        "I think I might call this thing 'telephone'.");
    photoPost.like();
    photoPost.like();
    photoPost.like();
    photoPost.like();
    NewsFeed feed = new NewsFeed();
    feed.addPost(messagePost);
    feed.addPost(photoPost);
    feed.show();
```

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

Network v3: dynamic dispatch

```
@Test
public void testShow() {
    MessagePost messagePost = new MessagePost("Leonardo da Vinci",
        "Had a great idea this morning.\n" +
        "But now I forgot what it was. Something to do with flying...");  

    messagePost.like();
    messagePost.like();
    PhotoPost photoPost = new PhotoPost("Alexander Graham Bell",
        "experiment.jpg",
        "I think I might call this thing 'telephone'.");
    photoPost.like();
    photoPost.like();
    photoPost.like();
    photoPost.like();
    NewsFeed feed = new NewsFeed();
    feed.addPost(messagePost);
    feed.addPost(photoPost);
    feed.show();
```

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

Output:

```
Had a great idea this morning.
But now I forgot what it was. Something to do with flying...
Leonardo da Vinci
0 seconds ago - 2 people like this.
No comments.

[experiment.jpg]
I think I might call this thing 'telephone'.
Alexander Graham Bell
0 seconds ago - 4 people like this.
No comments.
```

Network v3: dynamic dispatch

```
@Test  
public
```

```
MessagePost messagePost = new MessagePost("I think I might call this thing 'telephone'.");  
PhotoPost photoPost = new PhotoPost("experiment.jpg", "Had a great idea this morning.  
But now I forgot what it was. Something to do with flying...  
Leonardo da Vinci  
0 seconds ago - 2 people like this.  
No comments.  
  
[experiment.jpg]  
I think I might call this thing 'telephone'.  
Alexander Graham Bell  
0 seconds ago - 4 people like this.  
No comments.
```

Notar que, debido al dynamic dispatch `show` invoca primero a `display` de `MessagePost` (**primer objeto**), y luego a `display` de `PhotoPost` (**segundo objeto**)

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

Output:

Casting

- Cuando tenemos una variable polimórfica, no está permitido asignar un supertipo a un variable declarada de un subtipo
- Sin embargo, esta regla puede ser demasiado restrictiva en algunos casos
 - Por ejemplo, si analizamos la ejecución del siguiente programa sabemos que `v` tiene asignado un objeto de tipo `Car` en la última línea

```
Vehicle v;  
Car c = new Car();  
v = c; // correct  
c = v; // error
```

- Pero la asignación es rechazada por el compilador que sólo sabe que `v` es de tipo `Vehicle`
- Podemos indicarle al compilador que la variable `v` en tiempo de ejecución hace referencia a un objeto de tipo `Car`, forzando una conversión de tipos con el operador de casting

```
c = (Car) v; // okay
```

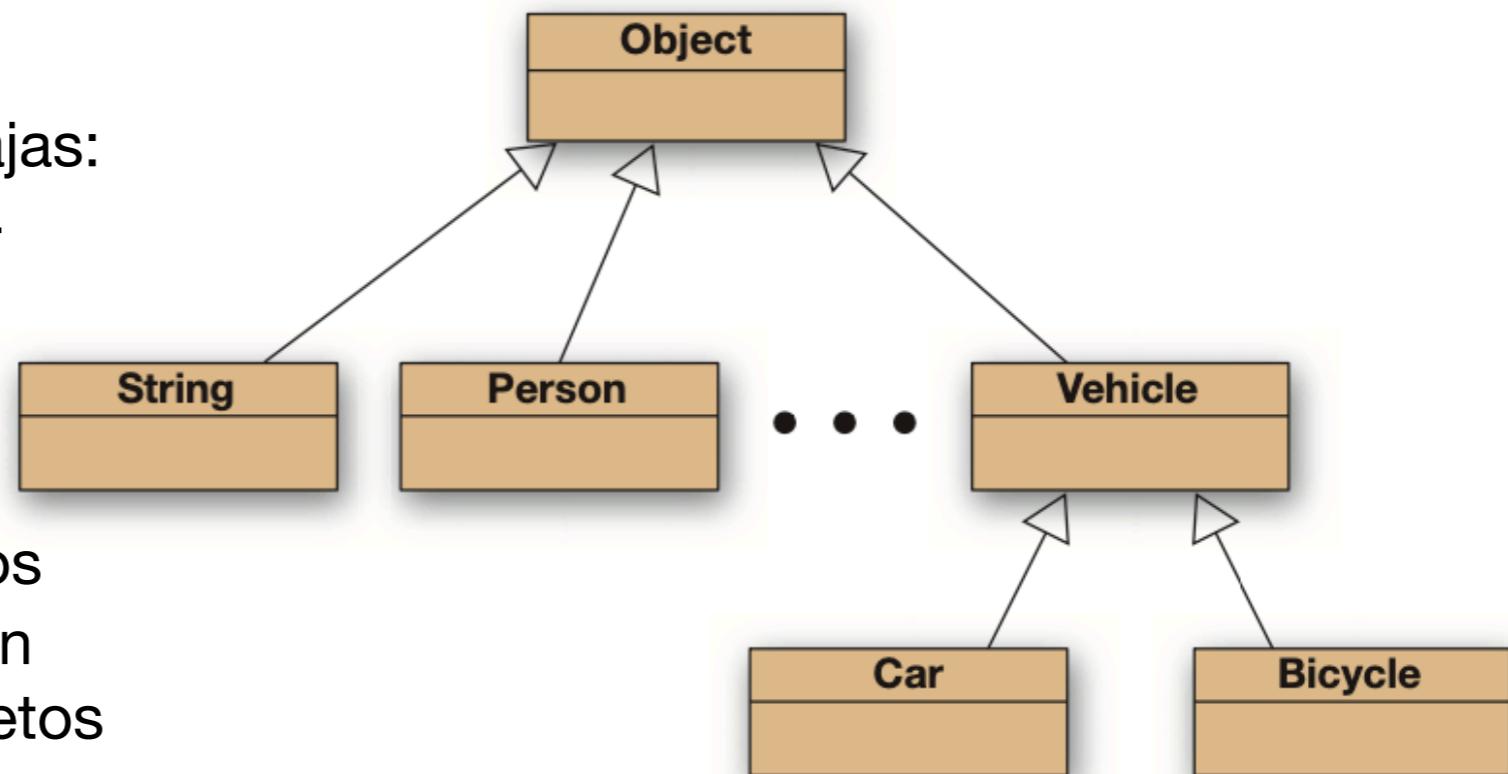
- Si en tiempo de ejecución `v` no almacena un objeto de tipo `Car`, la operación de casting falla, y puede provocar que el programa termine prematuramente

Casting

- Las operaciones de casting deben ser evitadas tanto como sea posible
 - Evitan que el compilador nos ayude a detectar errores de tipos en los programas
 - Pueden dar lugar a errores en tiempo de ejecución
- En la práctica, el casting rara vez es necesario si el programa está bien diseñado
- En casi todos los casos, es posible reestructurar el código para evitar las operaciones de casting

La clase Object

- En Java, todas las clases que no tienen una superclase declarada tienen a la clase `Object` como superclase
 - Es decir, `Object` es el supertipo de todos los objetos
- El compilador se encarga automáticamente de hacer que todas las clases extiendan de `Object`
- Que todos los objetos tengan una superclase común tiene dos ventajas:
 - Polimorfismo: Podemos definir variables polimórficas (de tipo `Object`), que almacenan objetos de cualquier tipo
 - La clase `Object` define algunos métodos importantes que están disponibles para todos los objetos
 - Ejemplos: `toString`, `equals` y `hashCode`



Métodos de Object: toString

- El método `toString` de `Object` devuelve una representación del objeto como un `String` [1]:

`String`

`toString()`

Returns a string representation of the object.

- Como convertir objetos a strings es una operación típica en muchas aplicaciones, la idea es usar siempre `toString` para esta operación, independientemente del tipo de objeto
- La versión por defecto implementada en `Object` no tiene mucha utilidad
 - Retorna un identificador para el objeto que incluye el nombre de la clase, seguido de un @ y un valor entero que representa al objeto (el `hashCode()` del objeto)

```
PhotoPost photoPost = new PhotoPost("Alexander Graham Bell",
    "experiment.jpg",
    "I think I might call this thing 'telephone'.");
System.out.println(photoPost.toString());
```

PhotoPost@65c221c0

- La utilidad de `toString` radica en que podemos redefinirlo para retornar representaciones de los objetos que tengan sentido para cada aplicación particular

Network v4: `toString`

- Vamos a refactorizar nuestro proyecto network para que los distintos tipos de posts implementen `toString()`
- De esta manera, el método `display` de la clase `Post` se simplifica significativamente:

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public void display()  
{  
    System.out.println(toString());  
}
```

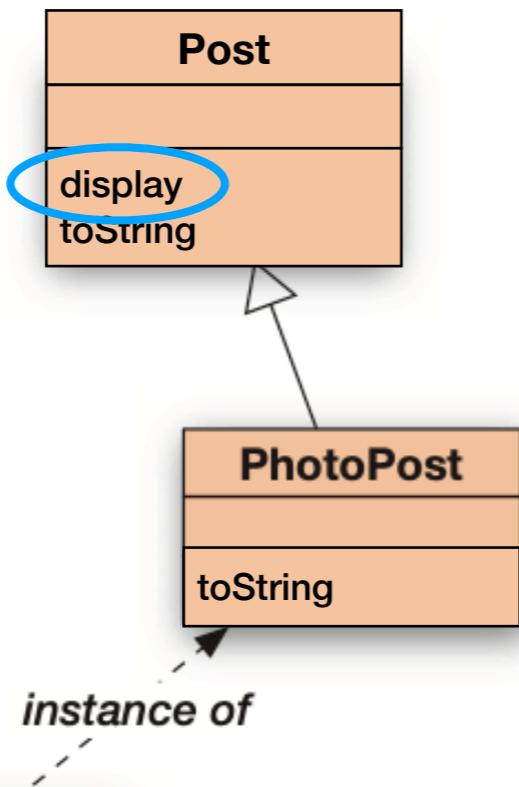
- Notar que no es necesario redefinir `display` en las subclases de `Post`:
 - Todas las subclases de `Post` implementarán `toString`, y el dynamic dispatch se encargará de llamar a la versión correcta de `toString`

Network v4: Dynamic dispatch

```
public void display()
{
    System.out.println(toString());
}
```

v1.display();

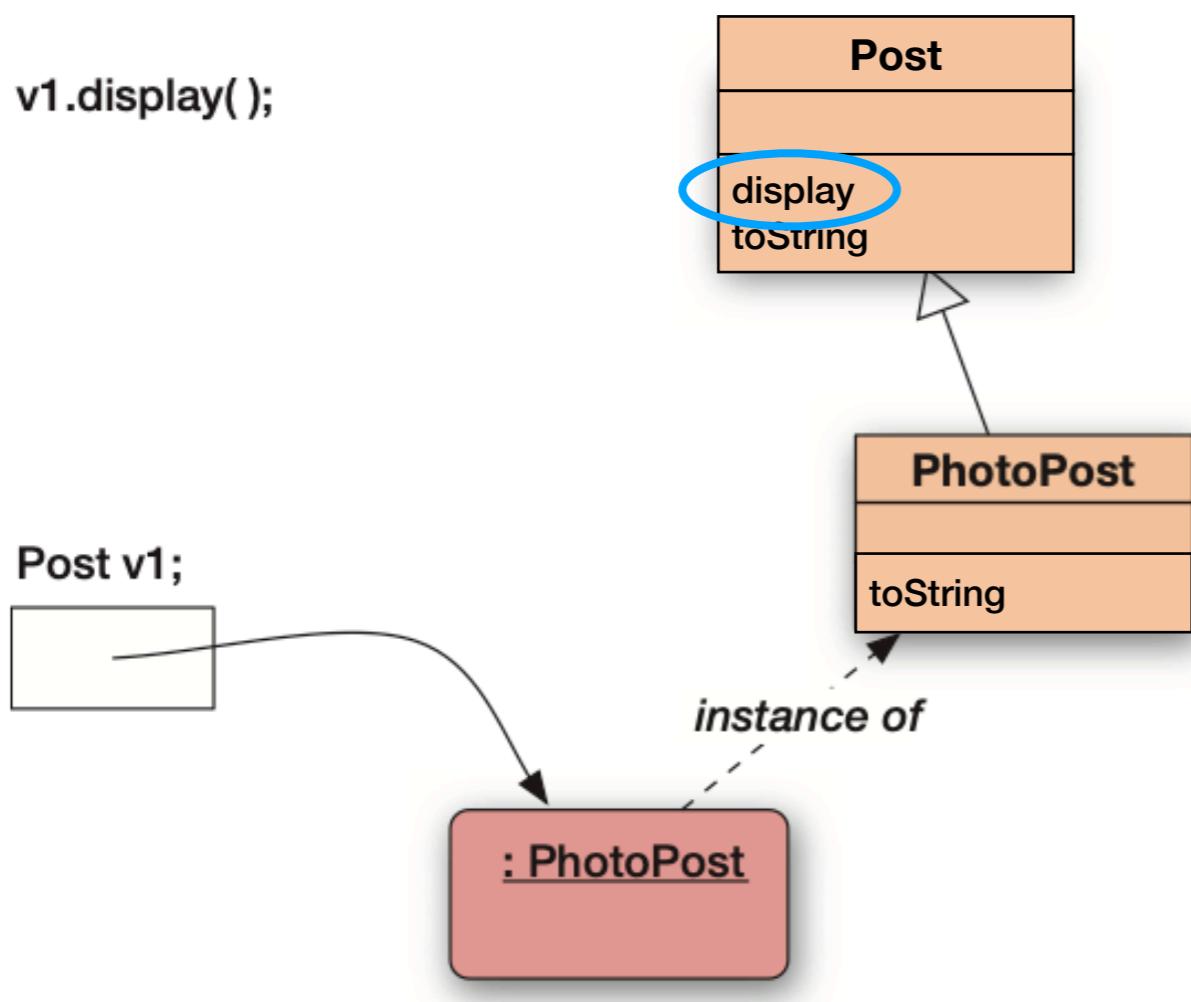
Post v1;



Network v4: Dynamic dispatch

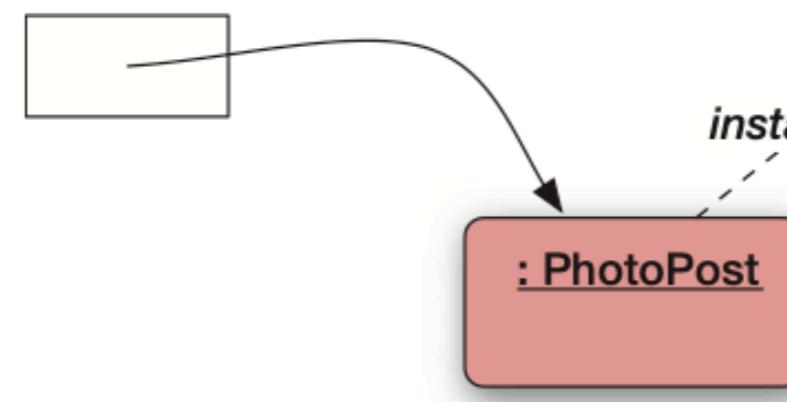
```
public void display()
{
    System.out.println(toString());
}
```

v1.display();



this.toString()

Post v1;



Network v4: toString

Clase Post

```
/**  
 * @post Return a String representation of the post.  
 */  
public String toString() {  
    String result = username + "\n";  
    result += timeString(timestamp);  
  
    if(likes > 0) {  
        result += " - " + likes + " people like this.\n";  
    }  
    else {  
        result += "\n";  
    }  
  
    if(comments.isEmpty()) {  
        result += " No comments.\n";  
    }  
    else {  
        result += " " + comments.size() + " comment(s). Click here to view.\n";  
    }  
    return result;  
}
```

Network v4: toString

Clase Post

```
/**  
 * @post Return a String representation of the post.  
 */  
public String toString() {  
    String result = username + "\n";  
    result += timeString(timestamp);  
  
    if(likes > 0) {  
        result += " - " + likes + " people like this.\n";  
    }  
    else {  
        result += "\n";  
    }  
  
    if(comments.isEmpty()) {  
        result += " No comments.\n";  
    }  
    else {  
        result += " " + comments.size() + " comment(s). Click here to view.\n";  
    }  
    return result;  
}
```

Clase MessagePost

```
/**  
 * @post Return a String representation of the post.  
 */  
public String toString() {  
    String result = message + "\n";  
    result += super.toString();  
    return result;  
}
```

Network v4: toString

Clase Post

```
/**  
 * @post Return a String representation of the post.  
 */  
public String toString() {  
    String result = username + "\n";  
    result += timeString(timestamp);  
  
    if(likes > 0) {  
        result += " - " + likes + " people like this.\n";  
    }  
    else {  
        result += "\n";  
    }  
  
    if(comments.isEmpty()) {  
        result += " No comments.\n";  
    }  
    else {  
        result += " " + comments.size() + " comment  
    }  
    return result;  
}
```

Clase MessagePost

```
/**  
 * @post Return a String representation of the post.  
 */  
public String toString() {  
    String result = message + "\n";  
    result += super.toString();  
    return result;  
}
```

Clase PhotoPost

```
/**  
 * @post Return a String representation of the post.  
 */  
public String toString() {  
    String result = "[" + filename + "]\n";  
    result += caption + "\n";  
    result += super.toString();  
    return result;  
}
```

Tests con `toString`

- El método `toString` puede aprovecharse para escribir aserciones para los test, como se muestra a continuación:

```
@Test
public void testToString() {
    PhotoPost photoPost = new PhotoPost("Alexander Graham Bell",
        "experiment.jpg",
        "I think I might call this thing 'telephone'.");
    String expected =
        "[experiment.jpg]\n" +
        "I think I might call this thing 'telephone'.\n" +
        "Alexander Graham Bell\n" +
        "0 seconds ago\n" +
        "  No comments.\n";
    assertTrue(expected.equals(photoPost.toString()));
}
```

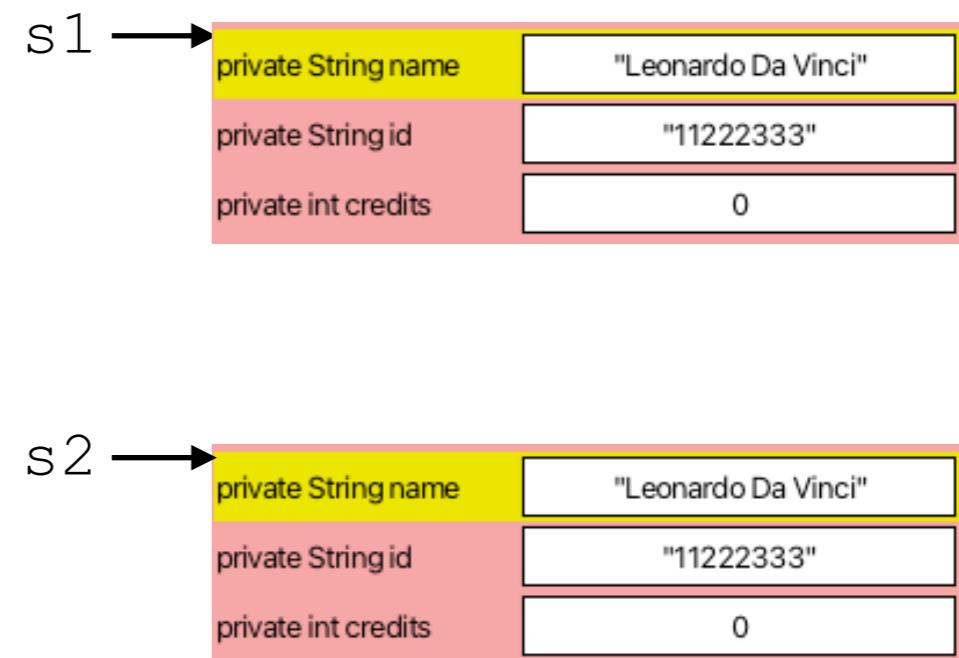
- Como vimos anteriormente, no es fácil testear lo que se imprime por terminal
- Sin embargo, podemos usar `toString` para testear una buena parte de la lógica de nuestra aplicación (menos lo que sale por terminal)

Igualdad de referencias

- Hay dos formas distintas de comparar objetos en Java
- Por un lado, podemos chequear que dos variables distintas hacen referencia al mismo objeto (al mismo espacio de memoria)
- El operador `==` verifica la igualdad de referencias

```
@Test
public void testReferenceEquality() {
    Student s1 = new Student("Leonardo Da Vinci", "11222333");
    Student s2 = new Student("Leonardo Da Vinci", "11222333");
    assertTrue(s1 == s1);
    assertTrue(s2 == s2);
    assertFalse(s1 == s2);
}
```

Diagrama de objetos:



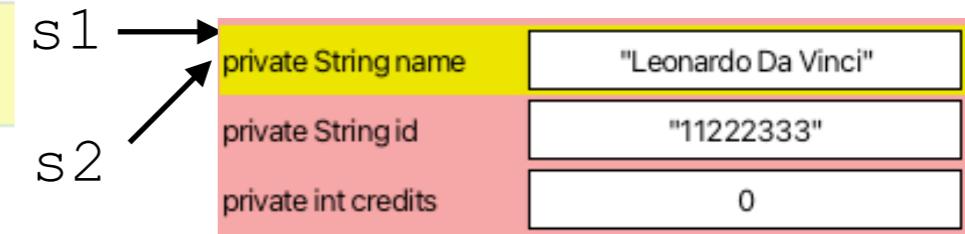
- En este caso, `s1` y `s2` hacen referencia a distintos objetos (aunque tengan exactamente el mismo contenido), y por lo tanto `s1 == s2` es **false**

Igualdad de referencias

- Para objetos, el operador de asignación cambia la referencia almacenada en la variable del lado izquierdo por una copia de la referencia de la expresión del lado derecho

Diagrama de objetos:

```
@Test
public void testAssignmentReferenceEquality() {
    Student s1 = new Student("Leonardo Da Vinci", "11222333");
    Student s2 = new Student("Leonardo Da Vinci", "11222333");
    s1 = s2;
    assertTrue(s1 == s2);
}
```



- Luego de la asignación `v1` y `v2` "apuntan" (guardan una referencia) al mismo objeto
- Los objetos que no son alcanzables desde ninguna variable del programa en ejecución son eliminadas periódicamente por el "garbage collector" de Java

garbage
collector

Métodos de Object: equals

- Para comparar la igualdad de objetos "por su contenido" usamos el método `equals` de la clase `Object`

`boolean`

`equals(Object obj)`

Indicates whether some other object is "equal to" this one.

- La versión por defecto del método, implementada en la clase `Object`, simplemente compara referencias

```
public boolean equals(Object obj)
{
    return this == obj;
}
```

- Redefinir el método `equals` nos da la posibilidad de definir una noción de igualdad de objetos que sea adecuada para cada aplicación particular

- Por ejemplo: Si sabemos que para el dominio de nuestra aplicación el campo `id` representa únicamente a un estudiante, podemos chequear igualdad de estudiantes mirando sólo este campo
- En caso contrario, podemos incluir algún otro campo adicional en el chequeo de igualdad, como nombre completo

Redefinición de equals

```
public class Student {  
    // the student's full name  
    private String name;  
    // the student ID  
    private String id;  
    // the amount of credits for student  
    private int credits;
```

```
    /**  
     * @post Returns true if and only if 'obj' is equal to 'this'.  
     */  
    public boolean equals(Object obj)  
    {  
        if(this == obj) {  
            return true; // Reference equality.  
        }  
        if(!(obj instanceof Student)) {  
            return false; // Not the same type.  
        }  
        // Gain access to the other student's fields.  
        Student other = (Student) obj;  
        return name.equals(other.name) &&  
               id.equals(other.id);  
    }
```

- Usualmente, lo primero que se chequea es si `this` y `obj` son la misma referencia
 - En este caso son el mismo objeto y se debe retornar `true`
- Luego, lo segundo que se verifica es que `obj` sea una instancia de la misma clase
 - Típicamente, si no son instancias de la misma clase se retorna `false`
- Finalmente, se chequea que los campos relevantes para la igualdad (en el dominio de aplicación particular) tengan el mismo valor

Redefinición de equals

```
@Test  
public void testObjectEquality() {  
    Student s1 = new Student("Leonardo Da Vinci", "11222333");  
    Student s2 = new Student("Leonardo Da Vinci", "11222333");  
    Student s3 = new Student("Alexander Graham Bell", "22333444");  
    assertFalse(s1 == s2);  
    assertTrue(s1.equals(s2));  
    assertFalse(s1.equals(s3));  
}
```

- Luego de redefinir equals en Student, s1.equals(s2) da verdadero porque los campos name e id de s1 y s2 tienen los mismos valores
 - Aunque s1 y s2 hacen referencia a objetos distintos (ocupan distintos espacios de memoria)

Diagrama de objetos:

| | | |
|------|---------------------|---------------------|
| s1 → | private String name | "Leonardo Da Vinci" |
| | private String id | "11222333" |
| | private int credits | 0 |

| | | |
|------|---------------------|---------------------|
| s2 → | private String name | "Leonardo Da Vinci" |
| | private String id | "11222333" |
| | private int credits | 0 |

El operador instanceof

- El operador `instanceof` nos permite verificar en tiempo de ejecución si el tipo de una variable es una clase dada:

```
obj instanceof MyClass
```

- Devuelve verdadero si `obj` es una instancia de la clase `MyClass`, y falso en caso contrario
- Por ejemplo, podemos obtener la lista de todos los posts que son instancias de `MessagePost` con el código a continuación:

```
ArrayList<MessagePost> messages = new ArrayList<>();
for(Post post : posts) {
    if(post instanceof MessagePost) {
        messages.add((MessagePost) post);
    }
}
```

equals y colecciones

- El método equals es muy importante para el funcionamiento de las colecciones en Java
- Veamos la especificación de algunos métodos de ArrayList:

```
public boolean contains(Object o)
```

Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (`o==null ? e==null : o.equals(e)`).

```
public boolean remove(Object o)
```

Removes the first occurrence of the specified element from this list, if it is present. If the list does not contain the element, it is unchanged. More formally, removes the element with the lowest index i such that (`o==null ? get(i)==null : o.equals(get(i))`) (if such an element exists). Returns true if this list contained the specified element (or equivalently, if this list changed as a result of the call).

```
public int indexOf(Object o)
```

Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. More formally, returns the lowest index i such that (`o==null ? get(i)==null : o.equals(get(i))`), or -1 if there is no such index.

equals y colecciones

- El método equals es muy importante para el funcionamiento de las colecciones en Java
- Veamos la especificación de algunos métodos de HashSet:

```
public boolean contains(Object o)
```

Returns true if this set contains the specified element. More formally, returns true if and only if this set contains an element e such that (`o==null ? e==null : o.equals(e)`).

```
public boolean add(E e)
```

Adds the specified element to this set if it is not already present. More formally, adds the specified element e to this set if this set contains no element e2 such that (`e==null ? e2==null : e.equals(e2)`). If this set already contains the element, the call leaves the set unchanged and returns false.

```
public boolean remove(Object o)
```

Removes the specified element from this set if it is present. More formally, removes an element e such that (`o==null ? e==null : o.equals(e)`), if this set contains such an element. Returns true if this set contained the element (or equivalently, if this set changed as a result of the call). (This set will not contain the element once the call returns.)

equals y colecciones

- El método equals es muy importante para el funcionamiento de las colecciones en Java
- Veamos la especificación de algunos métodos de HashSet:

```
public boolean contains(Object o)
```

Returns true if this set contains the specified element. More formally, returns true if and only if this set contains an element e such that (`o==null ? e==null : o.equals(e)`).

```
public boolean add(E e)
```

Adds the specified element to this set if it is not already present. More formally, adds the specified element e to this set if this set contains no element e2 such that (`e==null ? e2==null : e.equals(e2)`). If this set already

**Las implementaciones de colecciones en `java.util` usan el
método `equals` para comparar objetos**

```
public boolean remove(Object o)
```

Removes the specified element from this set if it is present. More formally, removes an element e such that (`o==null ? e==null : o.equals(e)`), if this set contains such an element. Returns true if this set contained the element (or equivalently, if this set changed as a result of the call). (This set will not contain the element once the call returns.)

equals y colecciones

- El método equals es muy importante para el funcionamiento de las colecciones en Java
- Veamos la especificación de algunos métodos de HashSet:

```
public boolean contains(Object o)
```

Returns true if this set contains the specified element. More formally, returns true if and only if this set contains an element e such that (`o==null ? e==null : o.equals(e)`).

```
public boolean add(E e)
```

Adds the specified element to this set if it is not already present. More formally, adds the specified element e to this set if this set contains no element e2 such that (`e==null ? e2==null : e.equals(e2)`). If this set already

**Las implementaciones de colecciones en `java.util` usan el
método `equals` para comparar objetos**

Removes the specified element from this set if it is present. More formally, removes an element e such that (`o==null ? e==null : o.equals(e)`) and contains(e). This method may not contain the element e, and if it does not, it does not affect the set.

Y sus propias implementaciones deberían hacer lo mismo, en caso contrario muchos métodos no van a funcionar correctamente

Métodos de Object: hashCode

- El método hashCode de Object genera un entero para representar el objeto al que se le aplica

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by [HashMap](#).

- Este método es requerido por colecciones implementadas con tablas de hash, como [HashMap](#) y [HashSet](#), para determinar eficientemente donde buscar/ubicar un objeto en la colección
 - Veremos las implementaciones de Maps y Sets con tablas de hashes más adelante

hashCode: Especificación implícita

- Por el momento, sólo es necesario cumplir con la especificación implícita de hashCode para que las aplicaciones que usan HashMap y HashSet funcionen correctamente:
 - Si dos objetos son iguales según equals, entonces deben tener el mismo hashCode
 - Formalmente: si `o1.equals(o2)` entonces `o1.hashCode() == o2.hashCode()`
 - Si esto no sucede, estas colecciones no funcionarán correctamente
 - Ej.: Un HashSet puede retornar que un elemento no pertenece al conjunto cuando si pertenece, o admitir elementos repetidos (según equals)
 - No se requiere que dos objetos distintos según equals tengan distintos hashCode's
 - Sin embargo, producir distintos enteros para objetos distintos puede mejorar significativamente la performance de estas estructuras
 - hashCode debe retornar siempre el mismo entero para el objeto durante una misma ejecución del programa, siempre que los atributos que se usan para comparar por igualdad el objeto no cambien
- Observación: Los tipos primitivos de Java, las clases wrapper, Strings, etc... ya implementan hashCode
 - Estos tipos pueden usarse con HashMap y HashSet sin problemas

Métodos de Object: hashCode

- Usaremos un método muy simple y efectivo para computar buenos `hashCodes` para nuestros objetos, tomado del libro Effective Java de Joshua Bloch [1]

1. Create a `int result` and assign a **non-zero** value.
2. For every field `f` tested in the `equals()` method, calculate a hash code `c` by:
 - If the field `f` is a `boolean`: calculate `(f ? 0 : 1)`;
 - If the field `f` is a `byte`, `char`, `short` or `int`: calculate `(int)f`;
 - If the field `f` is a `long`: calculate `(int)(f ^ (f >>> 32))`;
 - If the field `f` is a `float`: calculate `Float.floatToIntBits(f)`;
 - If the field `f` is a `double`: calculate `Double.doubleToLongBits(f)` and handle the return value like every long value;
 - If the field `f` is an *object*: Use the result of the `hashCode()` method or 0 if `f == null`;
 - If the field `f` is an *array*: see every field as separate element and calculate the hash value in a *recursive fashion* and combine the values as described next.

3. Combine the hash value `c` with `result`:

```
result = 37 * result + c
```

4. Return `result`

hashCode: Ejemplo

- Para la clase Student:
- Notar que el hashCode se calcula en base a los atributos observados para computar equals
- Algunos IDEs (ej. IntelliJ) generan automáticamente métodos equals y hashCode compatibles usando un algoritmo similar al anterior

```
/**  
 * @post Returns the hash code for object 'this'.  
 */  
public int hashCode()  
{  
    int result = 17; // An arbitrary starting value.  
    // Make the computed value depend on the order in which  
    // the fields are processed.  
    result = 37 * result + name.hashCode();  
    result = 37 * result + id.hashCode();  
    return result;  
}
```

```
public boolean equals(Object obj)  
{  
    if(this == obj) {  
        return true; // Reference equality.  
    }  
    if(!(obj instanceof Student)) {  
        return false; // Not the same type.  
    }  
    // Gain access to the other student's fields.  
    Student other = (Student) obj;  
    return name.equals(other.name) &&  
          id.equals(other.id);  
}
```

Clases abstractas

- Java permite la definición de clases abstractas: clases cuyo único propósito es ser superclases de otras clases
- No se pueden crear objetos a partir de las clases abstractas
 - El compilador de Java rechaza la creación de objetos para estas clases
- Para definir una clase como abstracta se agrega el modificador `abstract` en su encabezado
- Por ejemplo:

```
public abstract class Post
{
    private String username; // username of the post's author
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;
```

- Notar que es natural definir `Post` como clase abstracta, ya que esta clase no tiene la información suficiente como para representar un post real
 - Por lo tanto, no queremos permitir que se creen instancias de la clase `Post`

Métodos abstractos

- Las clases abstractas pueden definir métodos abstractos, aunque no están obligadas
- Los métodos abstractos definen el perfil y la especificación de un método, y no poseen implementación
- El objetivo es forzar que las subclases concretas de la clase abstracta implementen los métodos abstractos (respetando la especificación)
 - Si una clase concreta no implementa un método abstracto de una superclase el compilador da un error
- Un método se define abstracto agregando el modificador `abstract` en el perfil (y sin implementación)
- Por ejemplo, podríamos rediseñar nuestra clase `Post` para que el método `display` sea abstracto:

```
public abstract class Post
{
    [ . . . ]
    /**
     * @post Display the details of this post.
     *
     * (Currently: Print to the text terminal. This is simulating display
     * in a web browser for now.)
     */
    public abstract void display();
```

- Esto implica que las subclases de `Post` tienen la responsabilidad de implementar como se muestra cada tipo de post

Métodos abstractos

- Por ejemplo, Network v5 puede requerir que los distintos de post se muestren con distintos "marcos":

Clase Post

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public abstract void display();
```

Métodos abstractos

- Por ejemplo, Network v5 puede requerir que los distintos de post se muestren con distintos "marcos":

Clase Post

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public abstract void display();
```

Clase MessagePost

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public void display() {  
    System.out.println("-----");  
    System.out.println(toString());  
    System.out.println("-----");  
}
```

Métodos abstractos

- Por ejemplo, Network v5 puede requerir que los distintos de post se muestren con distintos "marcos":

Clase Post

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public abstract void display();
```

Clase MessagePost

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public void display() {  
    System.out.println("-----");  
    System.out.println(toString());  
    System.out.println("-----");  
}
```

Clase PhotoPost

```
/**  
 * @post Display the details of this post.  
 *  
 * (Currently: Print to the text terminal. This is simulating display  
 * in a web browser for now.)  
 */  
public void display() {  
    System.out.println("#####");  
    System.out.println(toString());  
    System.out.println("#####");  
}
```

Interfaces

- Una interfaz en Java (`interface`) es una especificación de un tipo de datos, en términos del nombre del tipo y un conjunto de métodos
 - Vamos a requerir especificaciones en las interfaces que expliquen como se usan los métodos del tipo
- Así, una interfaz es la especificación de una abstracción de datos
- Por ejemplo, en la figura vemos la interfaz para el tipo de datos `IntSet`
- Las interfaces no pueden tener constructores ni atributos
 - Sólo pueden tener métodos o atributos de clase (`static`), aunque no suelen usarse
- Todos los métodos de la interfaz deben ser públicos
- Para implementar una interfaz se usa en las subclases la palabra reservada `implements`

```
/*
 * IntSets are unbounded sets of integers.
 * A typical IntSet is {x1, . . . , xn}.
 */
public interface IntSet {

    /**
     * @post Adds 'x' to the elements of 'this',
     * i.e., this_post = this U { x }.
     */
    public void insert(int x);

    /**
     * @post Removes 'x' from 'this',
     * i.e., this_post = this \ { x }.
     */
    public void remove(int x);

    /**
     * @post If 'x' is in 'this' returns true else returns false.
     */
    public boolean isIn(int x);

    /**
     * @post Returns the cardinality of 'this'.
     * i.e., #this.
     */
    public int size();

    /**
     * @pre size() > 0
     * @post Returns an arbitrary element of 'this',
     * i.e., some e | this.isIn(e).
     */
    public int choose();
}
```

Ejemplo: java.util.List

java.util

Interface List<E>

<https://docs.oracle.com/javase/8/docs/api/java/util>List.html>

Type Parameters:

E - the type of elements in this list

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

boolean

add(E e)

Appends the specified element to the end of this list (optional operation).

void

add(int index, E element)

Inserts the specified element at the specified position in this list (optional operation).

void

clear()

Removes all of the elements from this list (optional operation).

boolean

contains(Object o)

Returns true if this list contains the specified element.

E

remove(int index)

Removes the element at the specified position in this list (optional operation).

boolean

remove(Object o)

Removes the first occurrence of the specified element from this list, if it is present (optional operation).

E

set(int index, E element)

Replaces the element at the specified position in this list with the specified element (optional operation).

int

size()

Returns the number of elements in this list.

Ejemplo: java.util.List

java.util

Interface List<E>

<https://docs.oracle.com/javase/8/docs/api/java/util>List.html>

Type Parameters:

E - the type of elements in this list

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

boolean

add(E e)

Appends the specified element to the end of this list (optional operation).

void

add(int index, E element)

Inserts the specified element at the specified position in this list (optional operation).

void

clear()

Removes all of the elements from this list (optional operation).

boolean

contains(Object o)

Returns true if this list contains the specified element.

E

remove(int index)

Removes the element at the specified position in this list (optional operation).

boolean

remove(Object o)

Removes the first occurrence of the specified element from this list, if it is present (optional operation).

E

set(int index, E element)

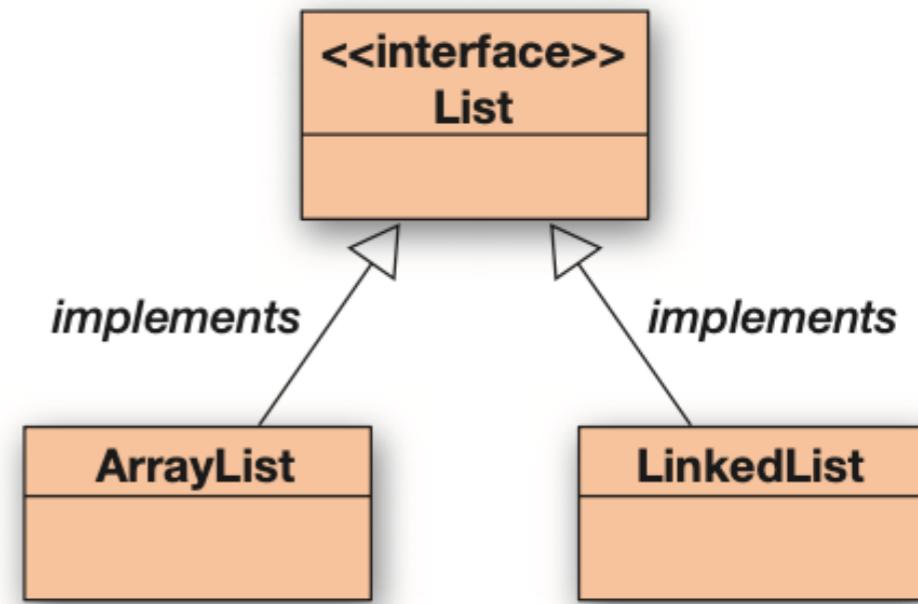
Replaces the element at the specified position in this list with the specified element (optional operation).

int

size()

Returns the number of elements in this list.

Dos implementaciones de List:



Programar respecto de abstracciones

- La ventaja principal de las interfaces es que nos permiten programar a los clientes respecto de abstracciones (la interfaz)
- Esto independiza el código del cliente de la implementación que elijamos del tipo
 - En otras palabras, desacopla al cliente de la implementación del tipo
- Si tenemos que cambiar la implementación de la abstracción para mejorar nuestro cliente, podemos modificar una única parte del código: la creación del objeto
 - Ej.: Para mejorar la eficiencia, para agregar nuevas operaciones, etc...
- Todo el código restante, si fue programado respecto de la abstracción (interfaz), se mantiene intacto
- Un principio de diseño que seguiremos es: programar respecto de abstracciones (tanto como sea posible)

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

    /**
     * @post Construct an empty news feed.
     */
    public NewsFeed()
    {
        posts = new ArrayList<>();
    }

    /**
     * @post Add a post to the news feed.
     */
    public void addPost(Post post)
    {
        posts.add(post);
    }

    /**
     * @post Show the news feed. Currently: print the news feed details
     * to the terminal. (To do: replace this later with display
     * in web browser.)
     */
    public void show()
    {
        // display all posts
        for(Post post : posts) {
            post.display();
            System.out.println(); // empty line between posts
        }
    }
}
```

Programar respecto de abstracciones

- La ventaja principal de las interfaces es que nos permiten programar a los clientes respecto de abstracciones (la interfaz)
- Esto independiza el código del cliente de la implementación que elijamos del tipo
 - En otras palabras, desacopla al cliente de la implementación del tipo
- Si tenemos que cambiar la implementación de la abstracción para mejorar nuestro cliente, podemos modificar una única parte del código: la creación del objeto
 - Ej.: Para mejorar la eficiencia, para agregar nuevas operaciones, etc...
- Todo el código restante, si fue programado respecto de la abstracción (interfaz), se mantiene intacto
- Un principio de diseño que seguiremos es: programar respecto de abstracciones (tanto como sea posible)

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

    /**
     * @post Construct an empty news feed.
     */
    public NewsFeed()
    {
        posts = new ArrayList<>();
    }

    /**
     * @post Add a post to the news feed.
     */
    public void addPost(Post post)
    {
        posts.add(post);
    }
}
```

NewsFeed programado respecto de la abstracción List:

```
classDiagram NewsFeed "NewsFeed" --> "List <<interface>>" ; NewsFeed --> "ArrayList <<interface>>" ; NewsFeed --> "LinkedList <<interface>>" ;
```

Programar respecto de abstracciones

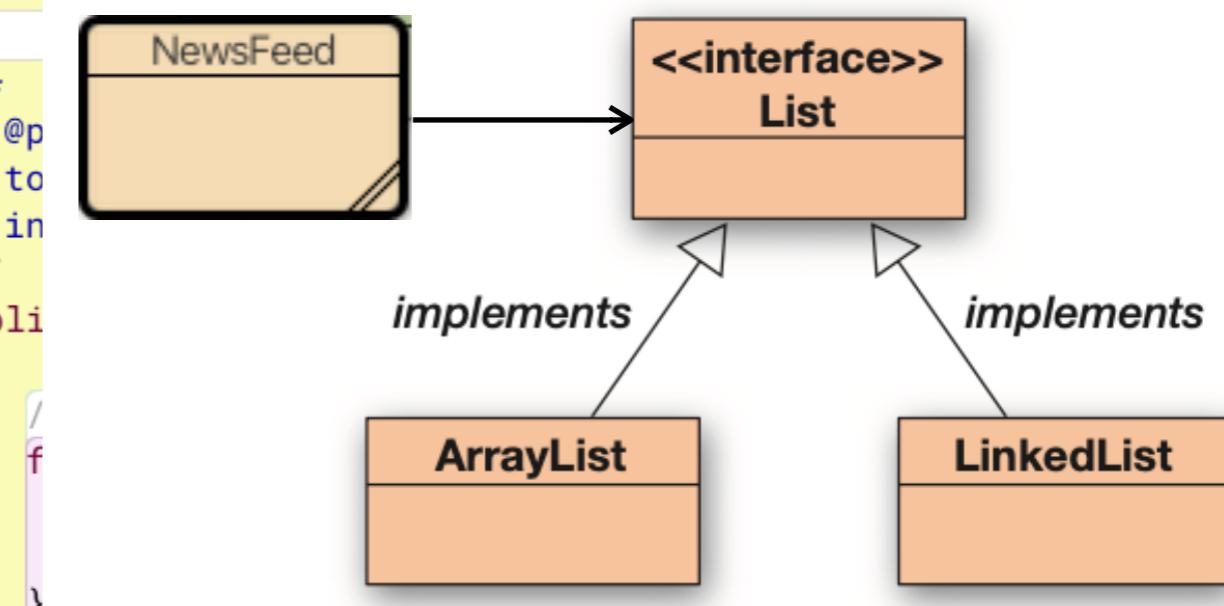
- La ventaja principal de las interfaces es que nos permiten programar a los clientes respecto de abstracciones (la interfaz)
- Esto independiza el código del cliente de la implementación que elijamos del tipo
 - En otras palabras, desacopla al cliente de la implementación del tipo
- Si tenemos que cambiar la implementación de la abstracción para mejorar nuestro cliente, podemos modificar una única parte del código: la creación del objeto
 - Ej.: Para mejorar la eficiencia, para agregar nuevas operaciones, etc...
- Todo el código restante, si fue programado respecto de la abstracción (interfaz), se mantiene intacto
- Un principio de diseño que seguiremos es: programar respecto de abstracciones (tanto como sea posible)

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

    /**
     * @post Construct an empty news feed.
     */
    public NewsFeed()
    {
        posts = new LinkedList<>();
    }

    /**
     * @post Add a post to the news feed.
     */
    public void addPost(Post post)
    {
        posts.add(post);
    }
}
```

NewsFeed programado respecto de la abstracción List:



Abstracciones de datos polimórficas

- Una de las ventajas del polimorfismo es que nos permite definir abstracciones de datos genéricas
- Por ejemplo, podemos definir colecciones que sirven para almacenar elementos de cualquier tipo
- Para esto, definimos las operaciones de manera que tomen como parámetros y retornen objetos del tipo `Object`
- Y almacenamos objetos de tipo `Object` en la colección
 - Como `Object` es supertipo de todos los tipos (de objetos), podemos almacenar objetos de cualquier tipo
- La desventaja de usar estas abstracciones polimórficas es que se pierden los chequeos de tipos
- Por error podríamos operar sobre una colección con objetos de un tipo distinto al tipo deseado de la colección
 - Ej.: Insertar enteros en una colección de Strings
- Para la comparación de igualdad de objetos típicamente se usa el método `equals` de `Object`
 - Hay que redefinir `equals` en los tipos que queremos almacenar para que la colección funcione correctamente

```
/*
 * Sets are unbounded sets of objects.
 * A typical Set is {o1, . . . , on}.
 * The methods use equals to determine
 * equality of elements.
 */
public interface Set {

    /**
     * @post Adds 'x' to the elements of 'this',
     * i.e., this_post = this U { x }.
     */
    public void insert(Object x);

    /**
     * @post Removes 'x' from 'this',
     * i.e., this_post = this \ { x }.
     */
    public void remove(Object x);

    /**
     * @post If 'x' is in 'this' returns true else returns false.
     */
    public boolean isIn(Object x);

    /**
     * @post Returns the cardinality of 'this'.
     * i.e., #this.
     */
    public int size();

    /**
     * @pre size() > 0
     * @post Returns an arbitrary element of 'this',
     * i.e., some e | this.isIn(e).
     */
    public Object choose();
}
```

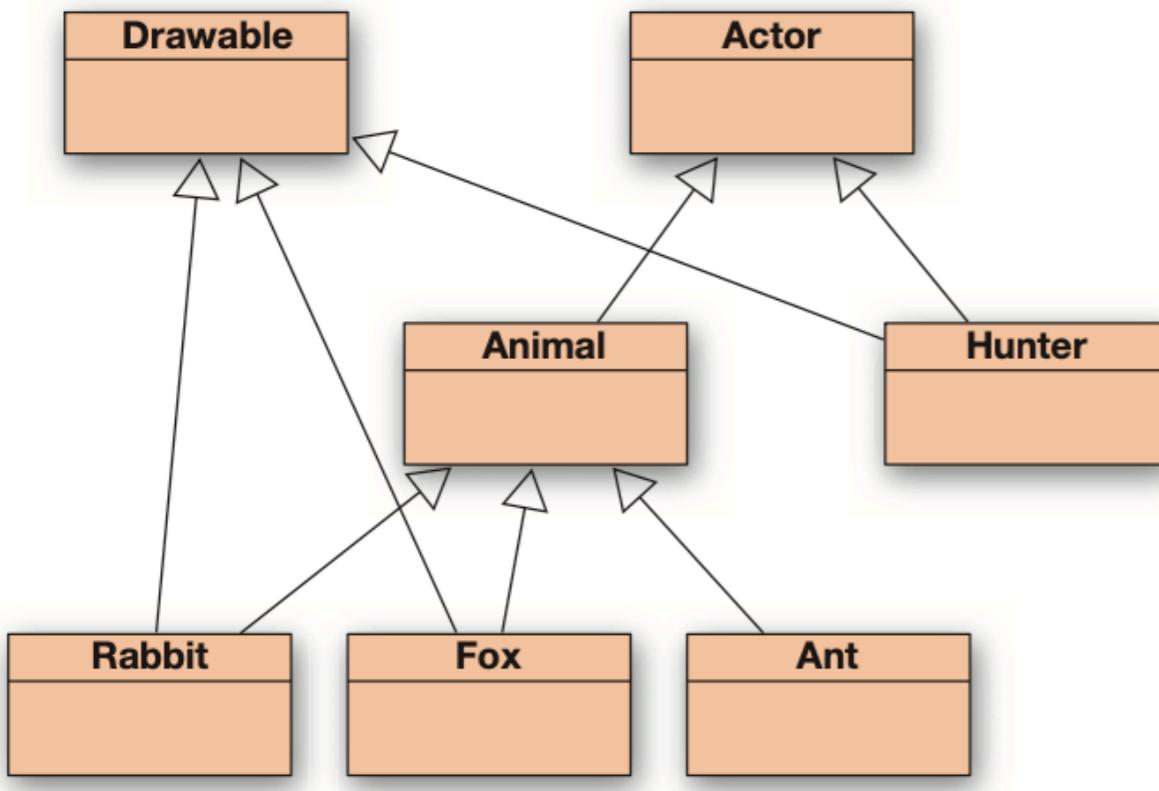
Genericidad

- En la mayoría de las aplicaciones, las colecciones son homogéneas
- Una mejor forma para definir abstracciones de datos genéricas es usando genericidad
- Por ejemplo, en la figura `Set<T>` indica que el conjunto almacena objetos de un tipo cualquiera `T`
 - Es decir, `Set<T>` define una familia de tipos
- `T` es una variable de tipos, y debe ser instanciada en la creación del objeto `Set` para definir el tipo real del conjunto
- Las operaciones y la implementación de la colección genérica se basan en la variable `T`
 - Ej.: Las operaciones `insert`, `remove` e `isIn` toman un objeto de tipo `T`, y `choose` retorna un objeto de tipo `T`
 - La implementación almacena objetos de tipo `T`
- Los objetos se crearán de manera similar a como creábamos colecciones de `java.util` (que están implementadas usando genericidad)
 - `Set<Integer> = new MySet<>();`
 - `Set<Person> = new MySet<>();`
- Con genericidad el compilador garantiza que los elementos usados para operar sobre la colección respetan el tipo de la colección

```
/**  
 * Sets are unbounded sets of objects of type T.  
 * A typical Set is {o1, . . . , on}.  
 * The methods use equals to determine equality of elements.  
 */  
public interface Set<T> {  
  
    /**  
     * @post Adds 'x' to the elements of 'this',  
     * i.e., this_post = this U { x }.  
     */  
    public void insert(T x);  
  
    /**  
     * @post Removes 'x' from 'this',  
     * i.e., this_post = this \ { x }.  
     */  
    public void remove(T x);  
  
    /**  
     * @post If 'x' is in 'this' returns true else returns false.  
     */  
    public boolean isIn(T x);  
  
    /**  
     * @post Returns the cardinality of 'this'.  
     * i.e., #this.  
     */  
    public int size();  
  
    /**  
     * @pre size() > 0  
     * @post Returns an arbitrary element of 'this',  
     * i.e., some e | this.isIn(e).  
     */  
    public T choose();  
}
```

Herencia múltiple

- Herencia múltiple describe la situación en la que una clase hereda de más de una superclase



```
public class Fox extends Animal implements Drawable
{
    Body of class omitted.
}
```

- Java no permite cualquier forma de herencia múltiple, e impone las siguientes restricciones
 - Una clase solo puede heredar de una única clase (`extends`)
 - Una clase puede implementar tantas interfaces como quiera (`implements`)
- Usando técnicas de diseño orientado a objetos estas limitaciones no traen mayores problemas en la práctica

Actividades

- Leer los capítulos 10, 11 y 12 del libro "Objects First with Java A Practical Introduction using BlueJ". Sixth Edition. D. Barnes & M. Kölking. Pearson. 2016

Bibliografía

- "Objects First with Java A Practical Introduction using BlueJ". Sixth Edition. D. Barnes & M. Kölking. Pearson. 2016
- "Program Development in Java - Abstraction, Specification, and Object-Oriented Design". B. Liskov & J. Guttag. Addison-Wesley. 2001
- Clean Code: A Handbook of Agile Software Craftsmanship (1st. ed.). Robert C. Martin. "Prentice Hall. 2008.