

Zlepšení struktury pomocí dědičnosti

Založeno na originální prezentaci ke Kapitole 8
„Improving structure with inheritance“ z učebnice
Objects First with Java - A Practical Introduction using
BlueJ, © David J. Barnes, Michael Kölling

Hlavní pojmy

- Dědičnost (Inheritance)
- Typy a podtypy
- Princip substituce
- Polymorfní proměnné

Projekt Network

- Prototyp malé části sociální sítě.
- Část, na kterou se zaměříme je kanál vybraných příspěvků – news feed.
- Jedná se o neustále aktualizovaný seznam novinek, které se zobrazují po otevření hlavní sít'ové stránky.
- Zkoumaný kód je zodpovědný za uchování a zobrazení příspěvků.

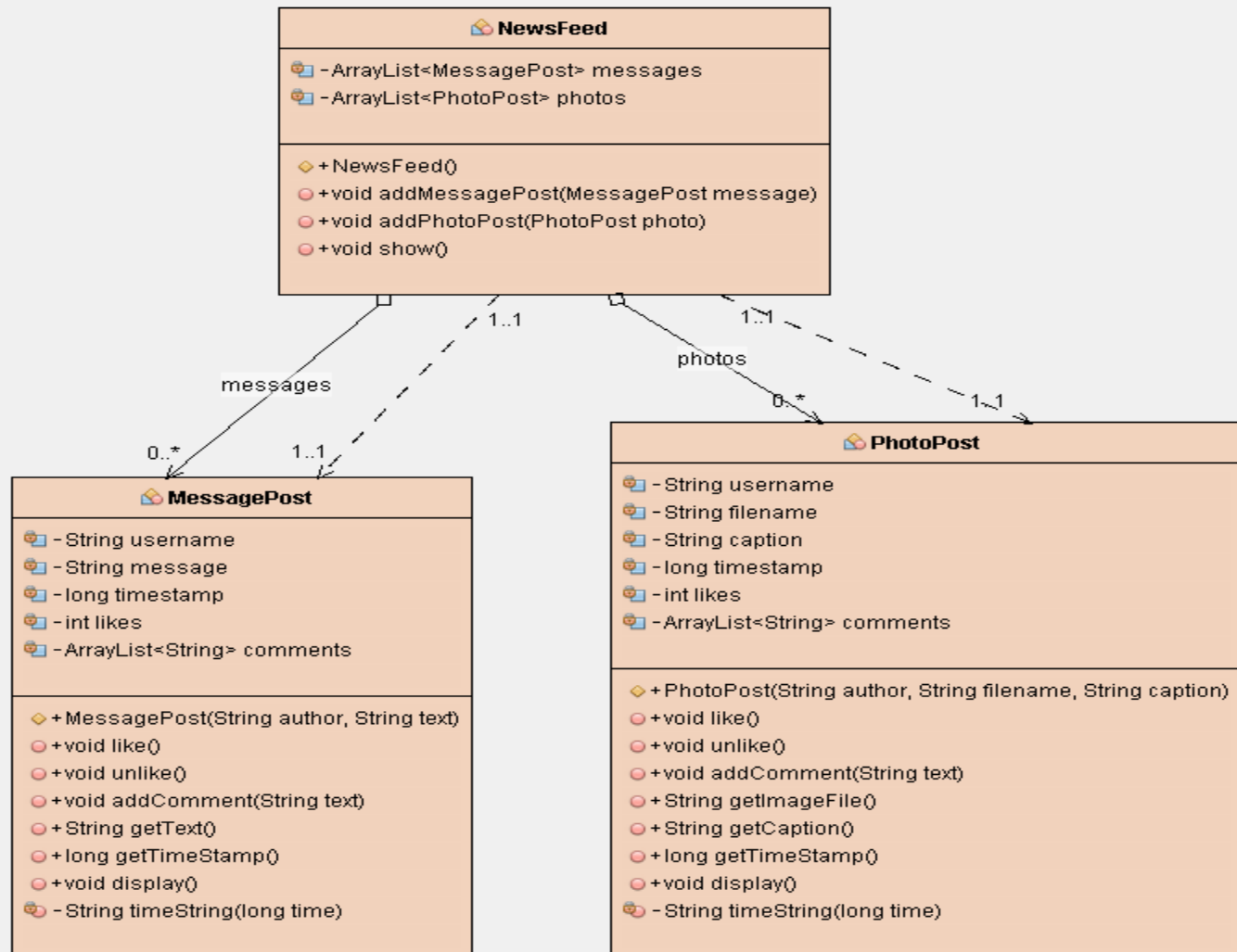
Projekt Network

- Uchovávané příspěvky, jejichž obsahem je text nebo obrázek budou instancemi dvou tříd.
 - **MessagePost**: víceřádkový textový příspěvek
 - **PhotoPost**: fotografie s titulkem
- Podporuje následující funkcionality:
 - vytvoření textových nebo fotografických příspěvků.

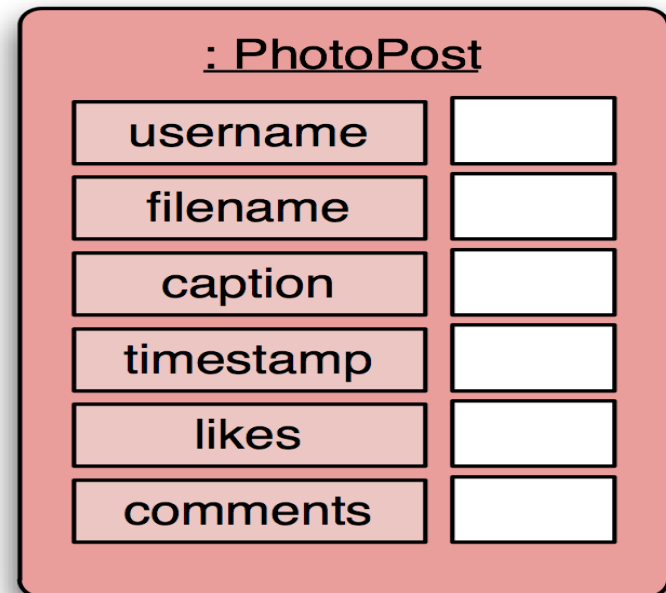
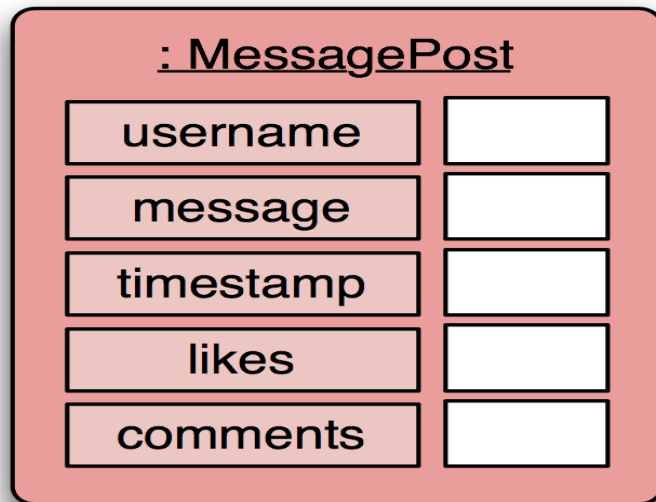
Projekt Network

- Textový příspěvek se skládá z několika řádků textu a fotografický příspěvek obsahuje foto s titulkem.
- Jsou uchovávány i další dodatečné informace.
- Vše uchováváno trvale pro pozdější použití.
- Nabízí se možnost:
 - vyhledávat v uchovaných příspěvcích,
 - zobrazit seznam příspěvků od jednoho uživatele nebo nejnovější od všech,
 - odstranit příspěvky.

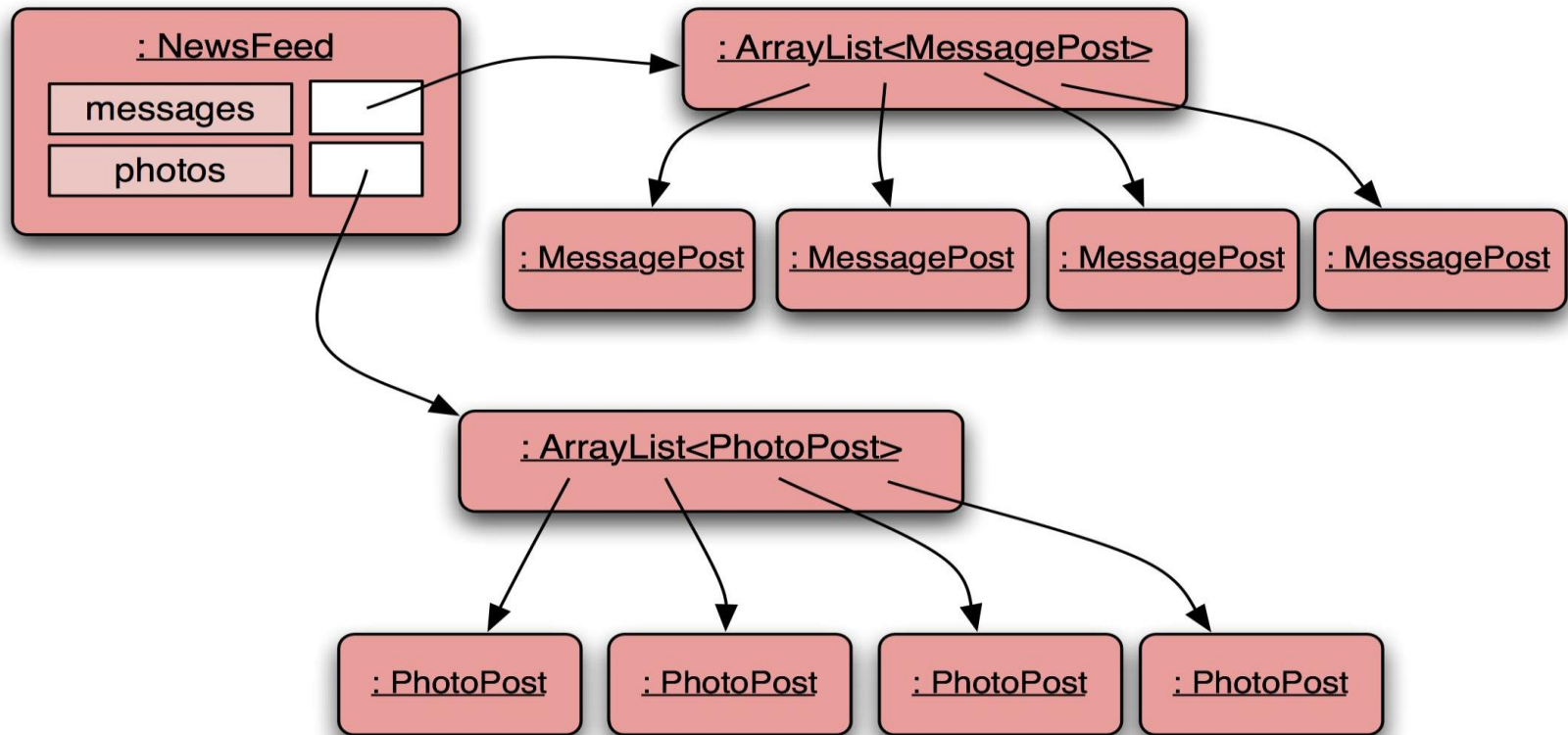
UML diagram tříd Network V1



Struktura instancí tříd MessagePost a PhotoPost v projektu Network V1



Struktura instancí třídy NewsFeed v projektu NetworkV1



Zdrojový kód třídy 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>();
    }

    public void addComment(String text)
    { ... }

    public void like()
    { ... }

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

Zdrojový kód třídy PhotoPost

```
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>();
    }

    public void addComment(String text)
    { ... }

    public void like()
    { ... }

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

Zdrojový kód třídy NewsFeed

```
public class NewsFeed
{
    private ArrayList<MessagePost> messages;
    private ArrayList<PhotoPost> photos;
    ...
    public void show()
    {
        for(MessagePost message : messages) {
            message.display();
            System.out.println(); // prázdný řádek mezi příspěvky
        }
        for(PhotoPost photo : photos) {
            photo.display();
            System.out.println(); // prázdný řádek mezi příspěvky
        }
    }
}
```

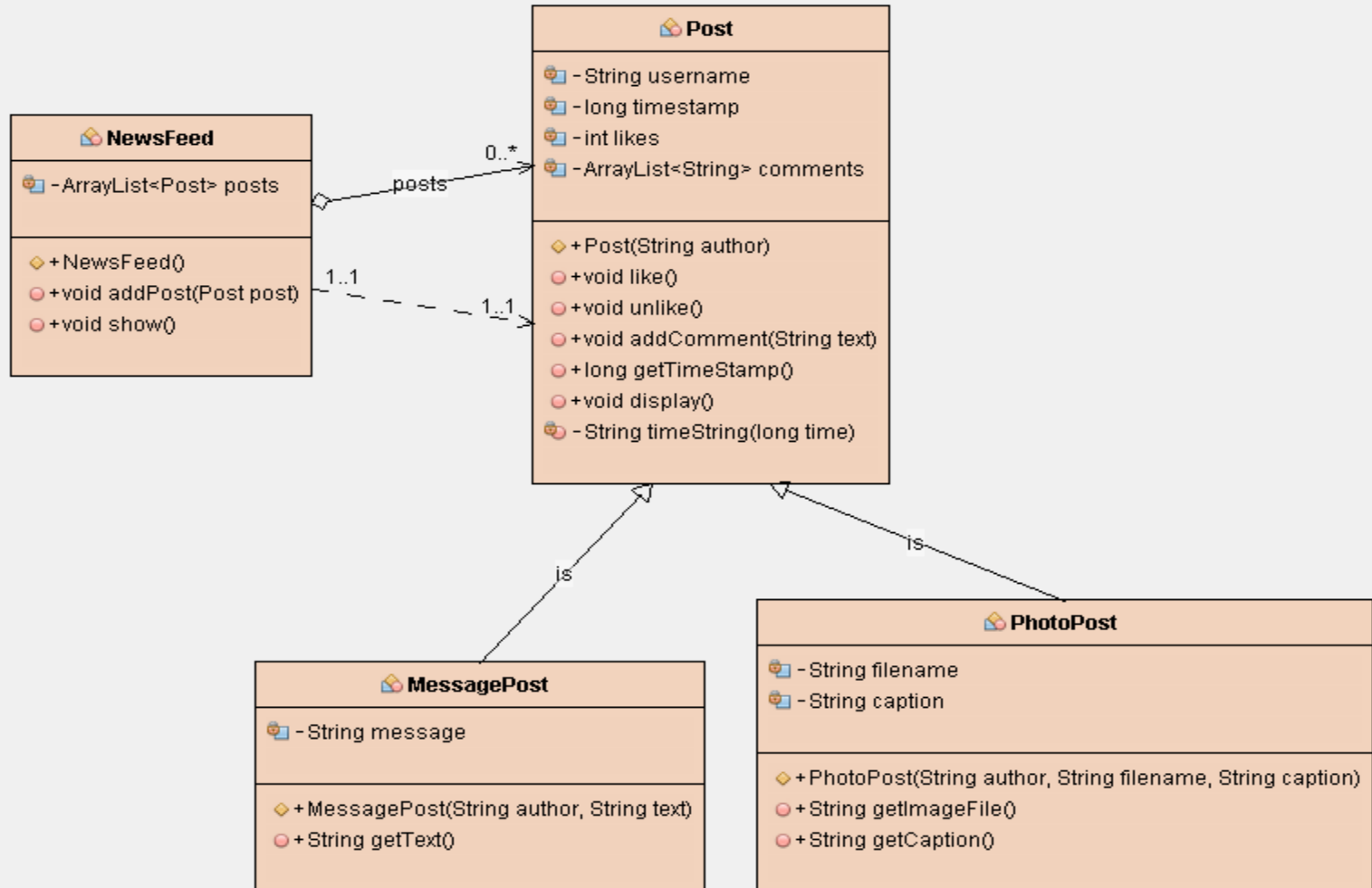
Kritika návrhu projektu Network

- Třídy **MessagePost** a **PhotoPost** jsou si velmi podobné (z velké části jsou identické).
- Duplicita kódu:
 - způsobuje, že provádění údržby je obtížnější a náročnější.
 - představuje nebezpečí chyb během nesprávně prováděné údržby.
- Duplicita kódu se vyskytuje i ve třídě **NewsFeed**.

Použití dědičnosti

- Definujeme jednu **nadtřidu**: **Post**
- Definujeme **podtřídy**: **MessagePost** a **PhotoPost**.
- Nadtřída definuje **společné atributy** (pomocí instančních proměnných) a **společné chování** (pomocí instančních metod).
- Podtřídy **dědí** atributy a chování z nadtřidy.
- Podtřídy **přidávají** vlastní atributy a vlastní metody.

UML diagram tříd Network V2



Dědičnost v jazyce Java

```
public class Post  
{  
    ...  
}
```

zde žádná změna

```
public class MessagePost  
extends Post  
{  
    ...  
}
```

```
public class PhotoPost  
extends Post  
{  
    ...  
}
```

změna zde

Nadtrída Post

```
public class Post
{
    private String userName;
    private long timeStamp;
    private int likes;
    private ArrayList<String> comments;

    // konstruktory a metody jsou vynechány.
}
```


Podtřídý

```
public class MessagePost extends Post
{
    private String message;
    // konstruktory a metody jsou vynechány.
}
```

```
public class PhotoPost extends Post
{
    private String fileName;
    private String caption;

    // konstruktory a metody jsou vynechány.
}
```

Dědičnost a konstruktory

```
public class Post
{
    private String userName;
    private long timeStamp;
    private int likes;
    private ArrayList<String> comments;

    /**
     * Inicializuje položky v objektu typu Post.
     */
    public Post(String author)
    {
        userName = author;
        timeStamp = System.currentTimeMillis();
        likes = 0;
        comment = new ArrayList<String>();
    }

    // metody jsou vynechány
}
```

Dědičnost a konstruktory

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

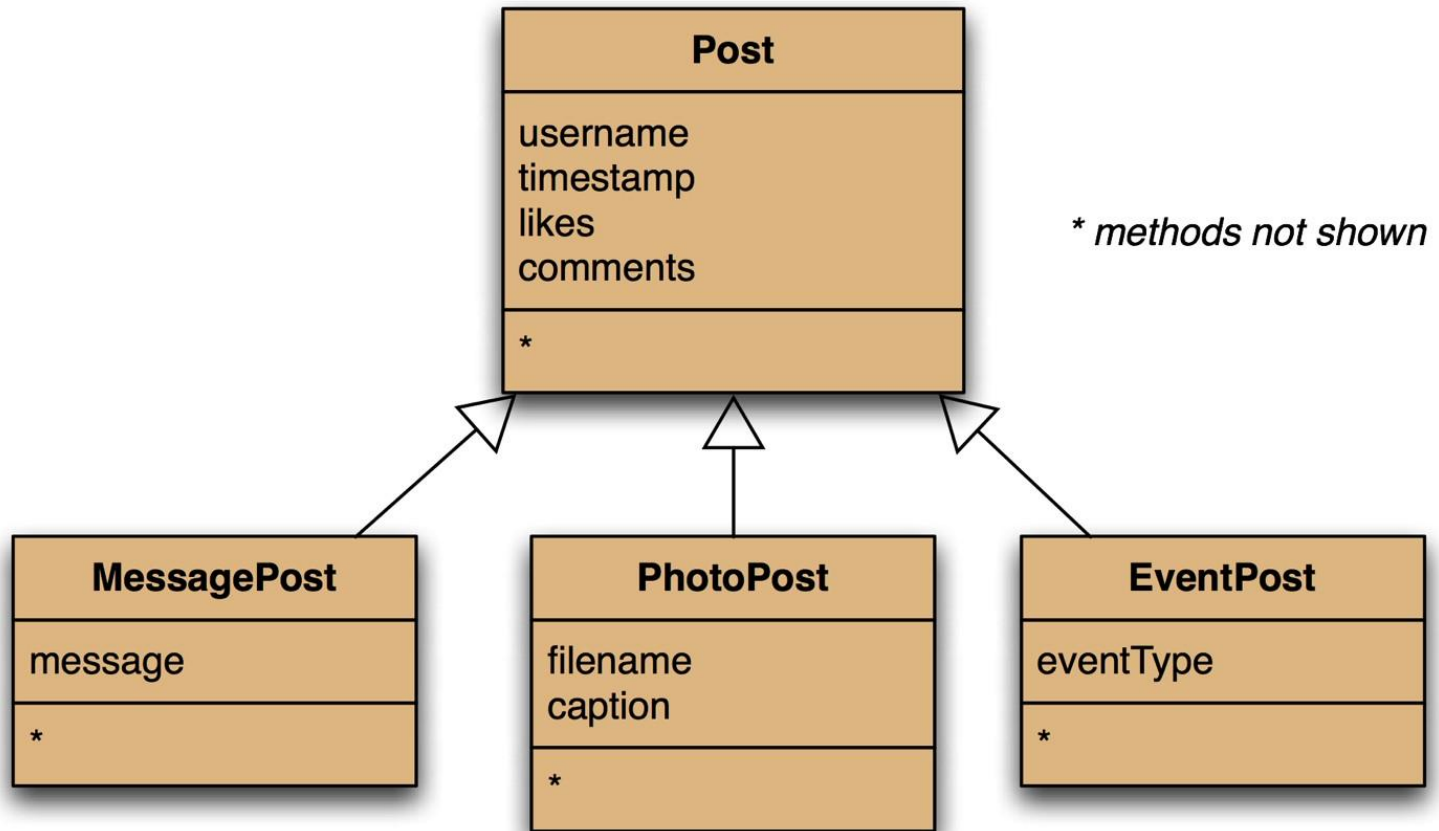
    /**
     * Konstruktor pro objekty třídy MessagePost
     */
    public MessagePost(String author, String text)
    {
        super(author) ;
        message = text;
    }

    // metody jsou vynechány
}
```

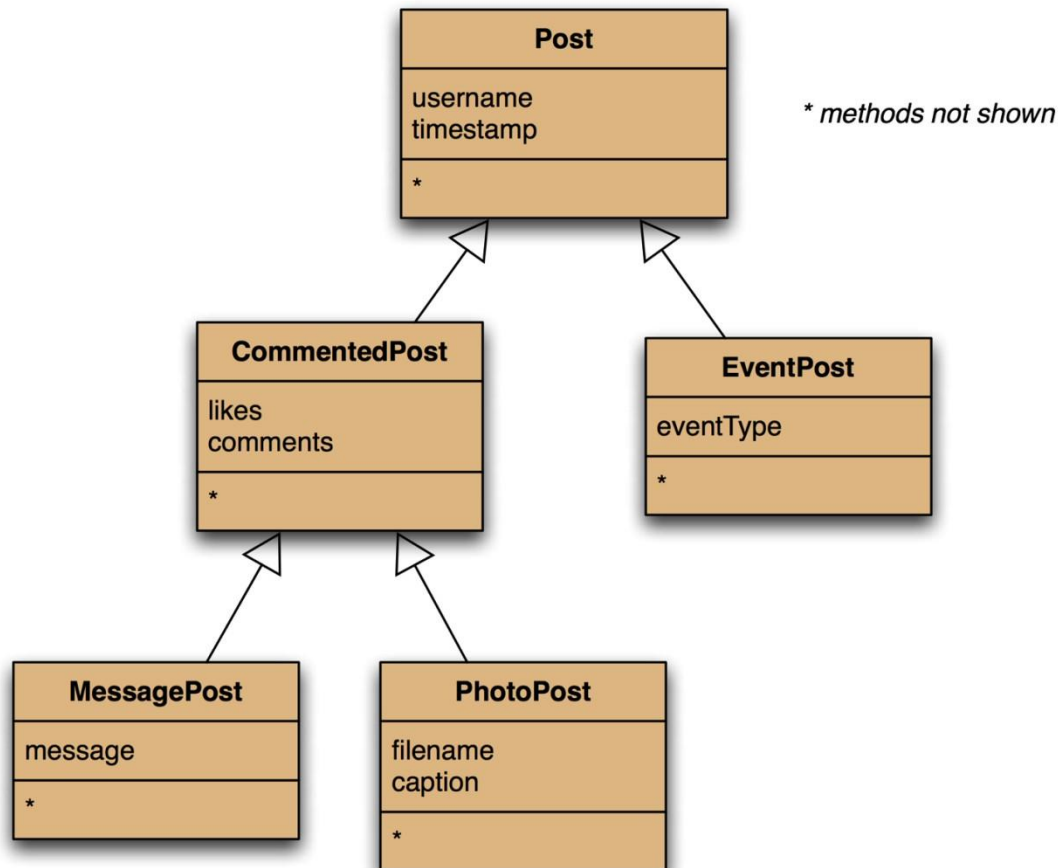
Volání konstruktoru nadtřídy

- Konstruktory podtřídy musí vždy obsahovat volání konstruktoru z nadtřídy.
- Jestliže není volání uvedeno, kompilátor vloží volání konstruktoru z nadtřídy (bez parametrů).
 - toto funguje pouze když nadtřída má konstruktor bez parametrů.
- Musí to být **první příkaz v konstruktoru** podtřídy.

Přidání dalších typů příspěvků



Hlubší hierarchie



Dosavadní přehled

Dědičnost pomáhá:

- vyhnout se duplicitě kódu
- opětovně využít kód
- se snadnějším udržováním
- s rozšiřováním kódu

Upravený zdrojový kód třídy NewsFeed

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

    /**
     * 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);
    }
    ...
}
```

Díky dědičnosti
jsme se zbavili
duplicity kódu ve
třídě **NewsFeed**.

Upravený zdrojový kód třídy NewsFeed

```
/**
 * Show the news feed. Currently: print the
 * news feed details to the terminal.
 * (Later: display in a web browser.)
 */
public void show()
{
    for(Post post : posts) {
        post.display();
        System.out.println(); // Empty line ...
    }
}
```

Podtypy

Nejprve jsme měli dvě metody:

```
public void addMessagePost(  
    MessagePost message)  
public void addPhotoPost(  
    PhotoPost photo)
```

Nyní máme pouze jednu metodu:

```
public void addPost(Post post)
```

Tuto metodu voláme takto:

```
PhotoPost myPhoto = new PhotoPost(...);  
feed.addPost(myPhoto);
```

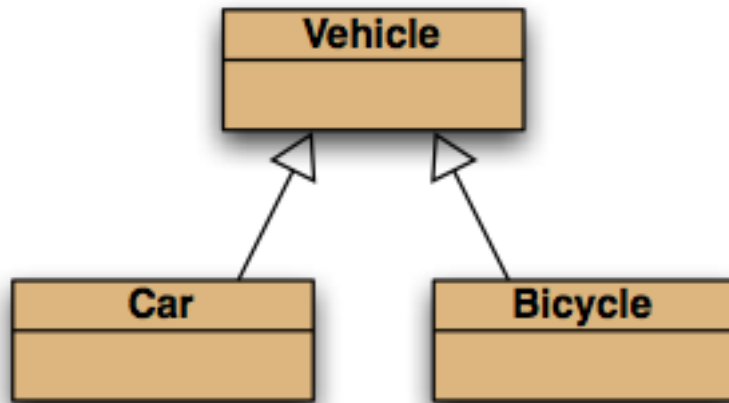
Podtřídy a podtypy

- Třídy definují **typy**.
- Podtřídy definují **podtypy**.
- Objekty podtříd mohou být použity tam, kde jsou vyžadovány objekty nadtřídy.
(Tomu se říká **princip substituce**.)

The Liskov Substitution Principle

- Barbara Liskov wrote LSP in 1988: "What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for **all** programs P defined in terms of T , the behavior of P is **unchanged** when o_1 is substituted for o_2 then S is a subtype of T ."
- [Barbara Liskov](#), Data Abstraction and Hierarchy, *SIGPLAN Notices*, 23,5 (May, 1988).
- Power of Abstraction
<https://www.youtube.com/watch?v=GDVAHA0oyJU>

Podtřídy a podtypy



**Objekty podtřídy třídy T
se mohou přiřadit
proměnné typu T.**

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

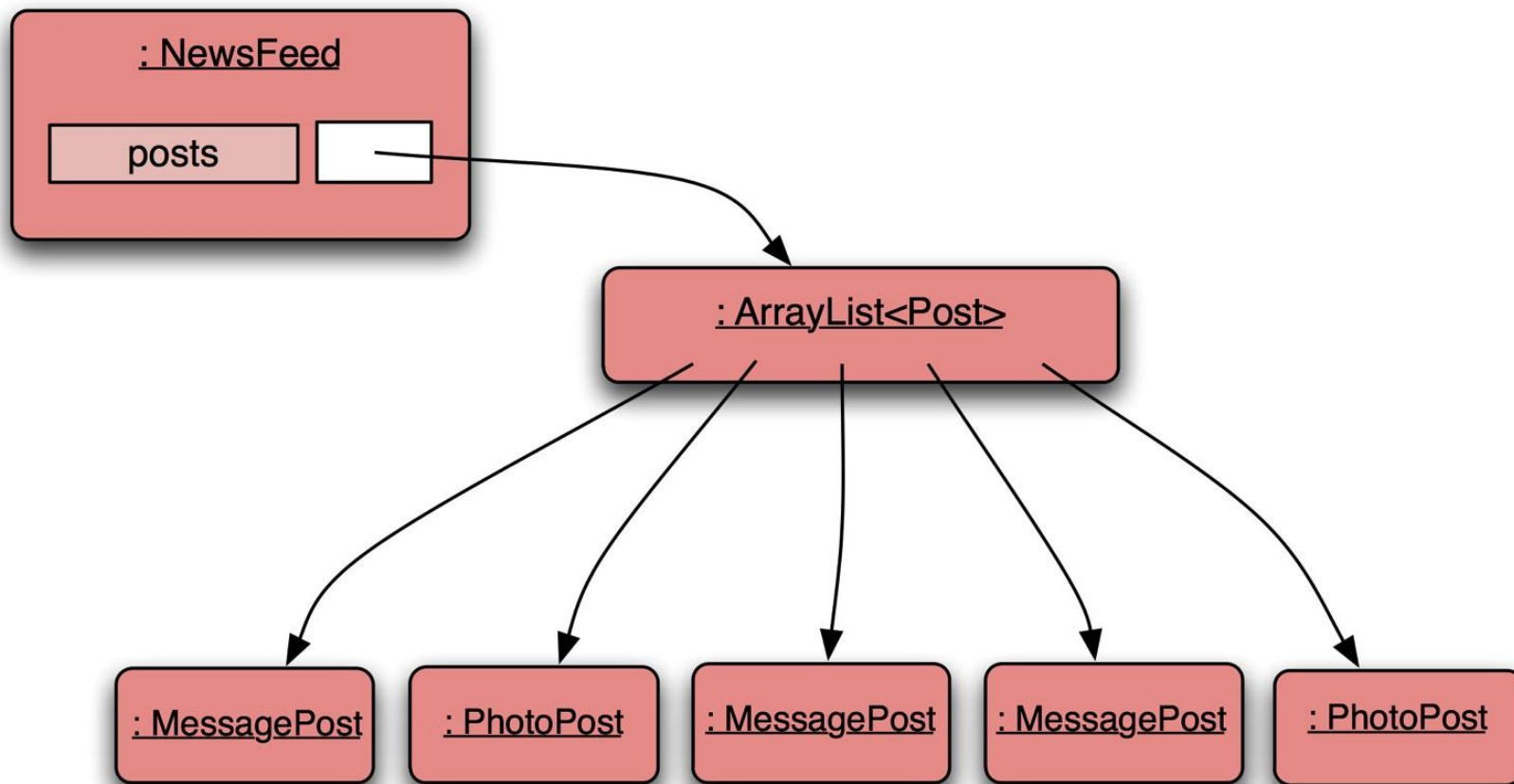
Podtypy a předávání parametrů

```
public class NewsFeed
{
    public void addPost(Post post)
    {
        ...
    }
}
```

Objekt podtřídy se může použít jako skutečný parametr odpovídající formálnímu parametru, jehož typ je dán nadtrídou.

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

Objektový diagram instancí třídy NewsFeed



Polymorfní proměnné

- Objektové proměnné jsou v jazyce Java **polymorfní**, protože mohou uchovávat objekty **více než jednoho typu**.
 - v jeden okamžik však logicky uchovávají vždy maximálně jeden objekt (odkaz na něj)
- Mohou uchovávat objekty deklarovaného typu nebo podtypů deklarovaného typu.

Přetypování

- Je možné přiřadit objekt podtypu typu T proměnné typu T.
- **Není možné přiřadit proměnné podtypu typu T objekt typu T!**

```
Vehicle v;  
Car c = new Car();  
v = c; // správně;  
c = v; // kompilační chyba!
```

- Přetypování to opraví:

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

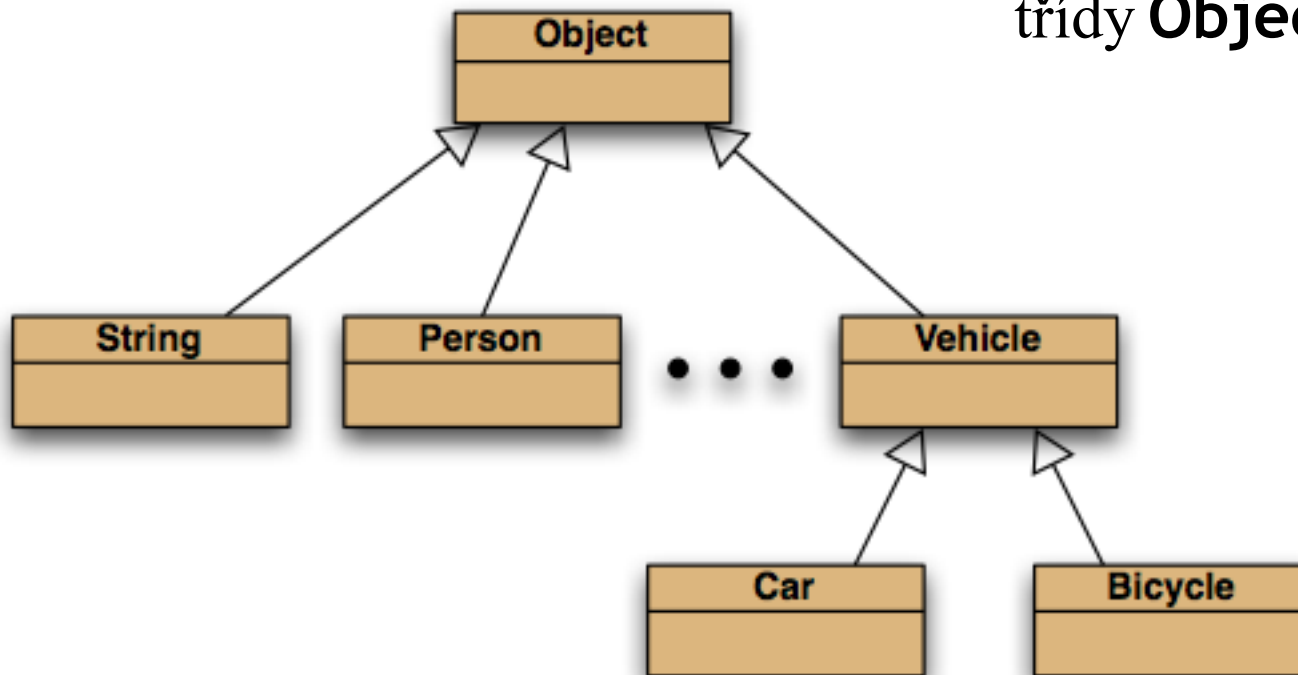
To je správně pouze v případě, že *v* je nějaká instance ze třídy **Car!**

Operátor přetypování

- Objektový typ v kulatých závorkách.
- Užívá se k překonání „ztráty typu“.
- Objekt není žádným způsobem měněn.
- Provádí se běhová kontrola pro zjištění, že objekt je opravdu toho typu:
 - **ClassCastException**, jestliže není!
- Užívat omezeně!

Třída Object

Všechny třídy dědí ze třídy **Object**.



Hierarchie tříd v balíčku java.util

java.lang.[Object](#) java.util.[AbstractCollection](#)<E> (implements java.util.[Collection](#)<E>)
java.util.[AbstractList](#)<E> (implements java.util.[List](#)<E>)
java.util.[AbstractSequentialList](#)<E>
java.util.[LinkedList](#)<E> (implements java.lang.[Cloneable](#), java.util.[Deque](#)<E>, java.util.[List](#)<E>, java.io.[Serializable](#))
java.util.[ArrayList](#)<E> (implements java.lang.[Cloneable](#), java.util.[List](#)<E>, java.util.[RandomAccess](#), java.io.[Serializable](#))
java.util.[Vector](#)<E> (implements java.lang.[Cloneable](#), java.util.[List](#)<E>, java.util.[RandomAccess](#), java.io.[Serializable](#))
java.util.[Stack](#)<E>
java.util.[AbstractQueue](#)<E> (implements java.util.[Queue](#)<E>)
java.util.[PriorityQueue](#)<E> (implements java.io.[Serializable](#))
java.util.[AbstractSet](#)<E> (implements java.util.[Set](#)<E>)
java.util.[EnumSet](#)<E> (implements java.lang.[Cloneable](#), java.io.[Serializable](#))
java.util.[HashSet](#)<E> (implements java.lang.[Cloneable](#), java.io.[Serializable](#), java.util.[Set](#)<E>)

Polymorfní kolekce

- Všechny kolekce jsou polymorfní.
- Prvky mohou být jednoduše typu **Object**.

```
public void add(Object element)
```

```
public Object get(int index)
```

- Preferováno je generické použití kolekcí!

```
př. ArrayList<E>
```

```
public void add(E element)
```

```
public E get(int index)
```

Polymorfní kolekce

- Typový parametr **omezuje stupeň polymorfismu.**

`ArrayList<Post>`

- Metody pro práci s kolekcemi jsou typovány.
- Bez typového parametru použije kompilátor `ArrayList<Object>`, což způsobí pravděpodobně při kompilaci **varování “unchecked or unsafe operations”**.

Častější nutnost použít přetypování.

Kolekce a primitivní typy

- Do kolekcí lze vkládat objekty všech typů...
- ... protože kolekce akceptují objekty typu `Object` (viz `add(Object o)`)
- ... a typy všech tříd jsou podtypy typu `Object`.
- Ohromné! Ale co primitivní typy?

Obalové třídy

- Primitivní typy (`int`, `char`, atd.) nejsou objektové typy. Hodnoty primitivních typů musí být zabaleny do objektů!
- Obalové třídy existují pro všechny primitivní typy:

primitivní typ

`int`

`float`

`char`

...

obalová třída

`Integer`

`Float`

`Character`

...

Obalové třídy

```
int i = 18;  
Integer iwrap = new Integer(i);  
...  
int value = iwrap.intValue();
```

← zabal hodnotu

← vybal hodnotu

**Autoboxing a unboxing
v praxi umožňuje, že toto nemusíme
explicitě dělat**

Autoboxing a unboxing

```
private ArrayList<Integer> markList;
```

```
...
```

```
public void storeMark(int mark)
```

```
{
```

```
    markList.add(mark); autoboxing
```

```
}
```

```
int firstMark = markList.remove(0); unboxing
```

Přehled

- Dědičnost umožňuje definovat třídy jako rozšíření jiných tříd.
- Dědičnost
 - umožňuje se vyhnout duplicitě kódu
 - umožňuje opakované využití kódu
 - zjednodušuje kód
 - zjednodušuje udržování a rozšiřování kódu
- Proměnné mohou uchovávat objekty podtypu typu proměnné.
- Objekty podtypu typu T mohou být použity kdekoliv, kde se očekávají objekty typu T (princip substituce).