

# Více o dědičnosti

## Zkoumání polymorfismu

Založeno na originální prezentaci ke Kapitole 9

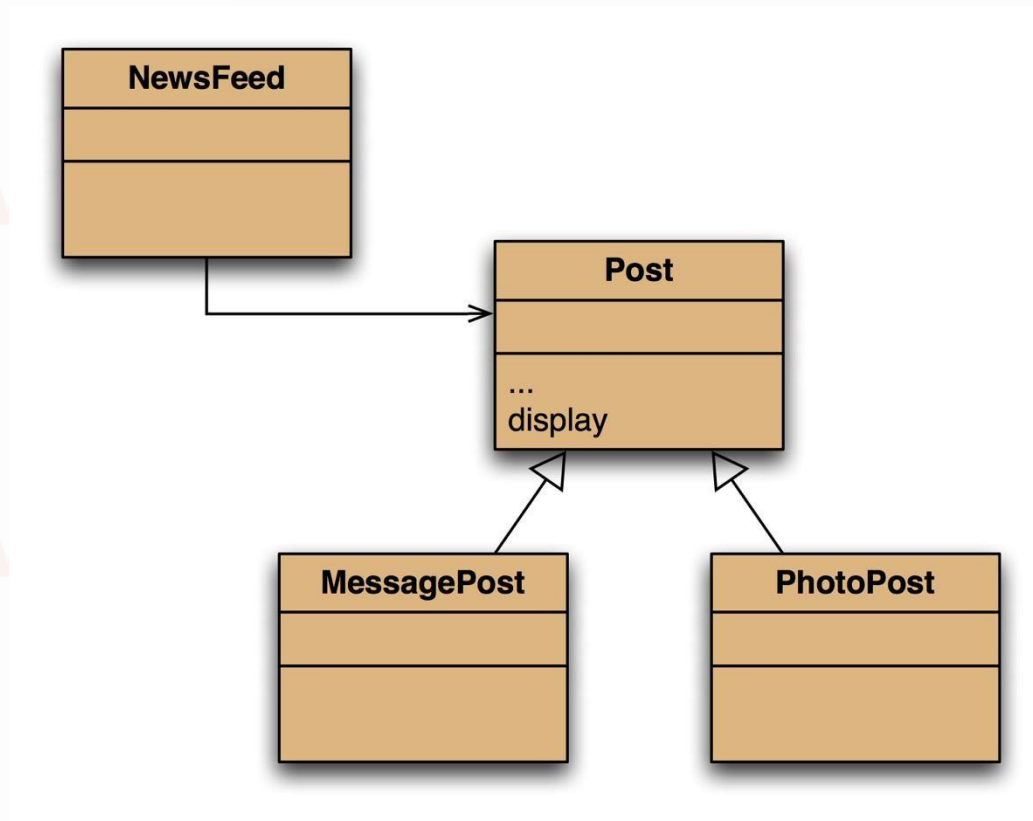
„More about inheritance“ z učebnice

Objects First with Java - A Practical Introduction using  
BlueJ, © David J. Barnes, Michael Kölling

# Hlavní pojmy

- polymorfismus metod
- statický a dynamický typ
- překrývání (overriding)
- dynamické vyhledávání metody
- přístup typu protected

# Hierarchie tříd v projektu Network V2



# Rozporný výstup

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.

**To co chceme**

Leonardo da Vinci

40 seconds ago - 2 people like this.

No comments.

Alexander Graham Bell

12 minutes ago - 4 people like this.

No comments.

**To co máme**

# Metoda display

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

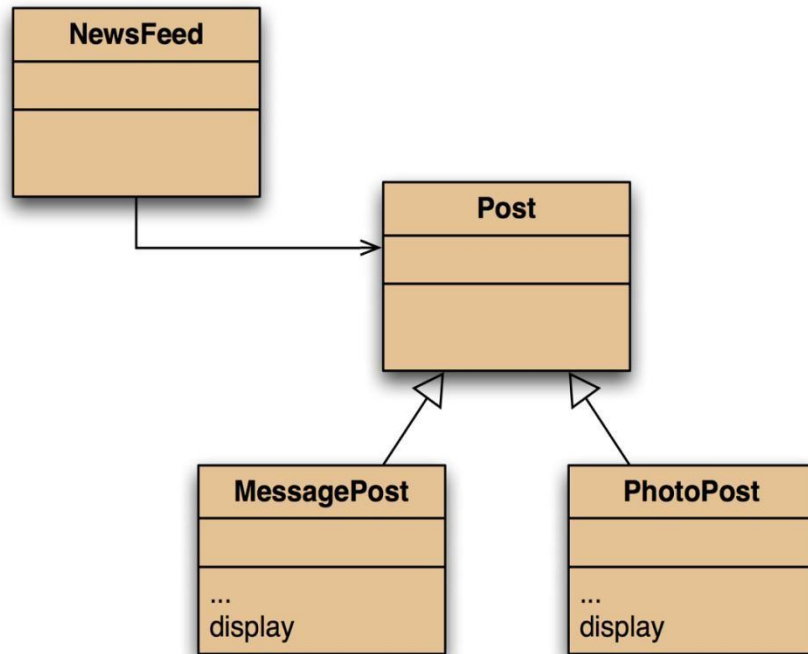
    if (likes > 0) {
        System.out.println("  - " + likes + " people like this.");
    } else {
        System.out.println();
    }

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

# Problém

- Metoda **display** ve třídě **Post** vypisuje pouze společné položky.
- Dědičnost je jednosměrná ulice:
  - Podtřída dědí položky z nadtříd.
  - Nadtřída neví nic o položkách její podtříd.

# Pokus o řešení problému



- Umístíme metodu **display** tam, kde „má přístup k informacím“, které potřebuje.
- Každá podtřída má svoji vlastní verzi.
- Ale položky ve třídě **Post** mají privátní přístup.
- **NewsFeed** nemůže nalézt metodu **display** ve třídě **Post**.

# Statický a dynamický typ

- Složitější typová hierarchie vyžaduje k popisu další pojmy.
- Nová terminologie:
  - statický typ
  - dynamický typ
  - vyhledání metody (*method lookup, method binding or method dispatch*)

# Statický a dynamický typ

Jaký je typ c1?

```
Car c1 = new Car();
```

Jaký je typ v1?

```
Vehicle v1 = new Car();
```

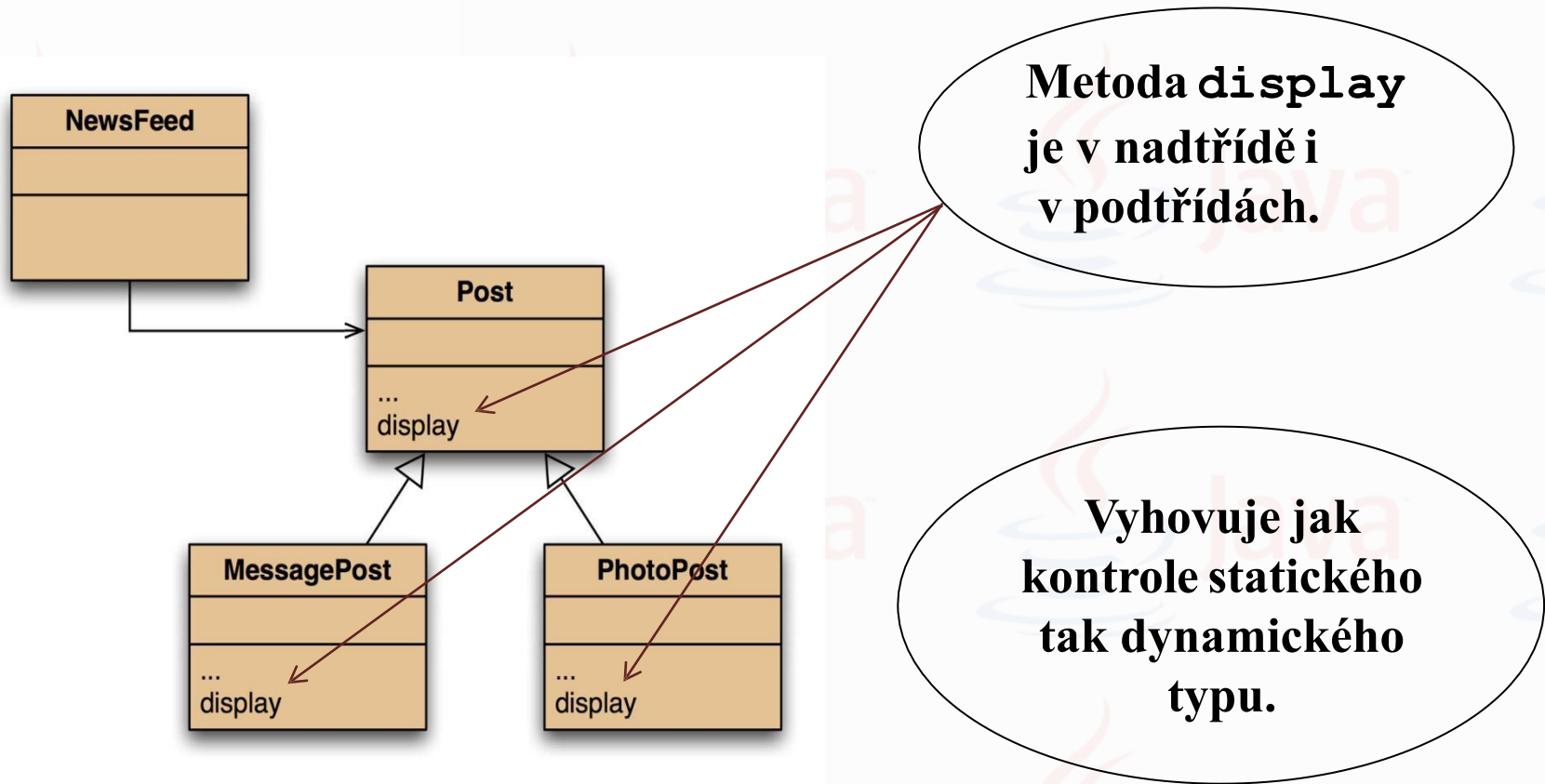
# Statický a dynamický typ

- Deklarovaný typ proměnné je její *statický typ*.
- Typ objektu, který proměnná referencuje je její *dynamický typ*.
- Úlohou kompilátoru je kontrolovat narušení *statického typu*.

```
for(Post item : items) {  
    item.display();  
}
```

**// Kompilační chyba,  
metoda display ve třídě  
Post chybí.**

# Překrývání metod je řešení



# Překrývání (Overriding) instančních metod

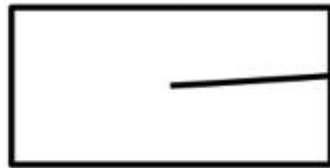
- Nadtřída i podtřída deklarují metody *se stejnou signaturou* (včetně návratového typu).
- Každá má přístup k položkám své třídy.
- Nadtřída uspokojuje kontrolu založenou na statických typech.
- Metoda podtřídy je volána při běhu –*překrývá* verzi metody z nadtřídy.
- Co se stane s verzí z nadtřídy?

# Překrývání (Overriding) instančních metod

- Překrývající metoda může vrátit podtyp typu, který vrací překrývaná metoda- *kovariantní návratový typ*.
- Překrývající metodu lze označit anotací **@Override**.
  - Kontrola existence metody se stejnou signaturou v nadtřídě.

# Rozdílné statické a dynamické typy

Post post



: PhotoPost



# Vyhledávání metody

`v1.display();`

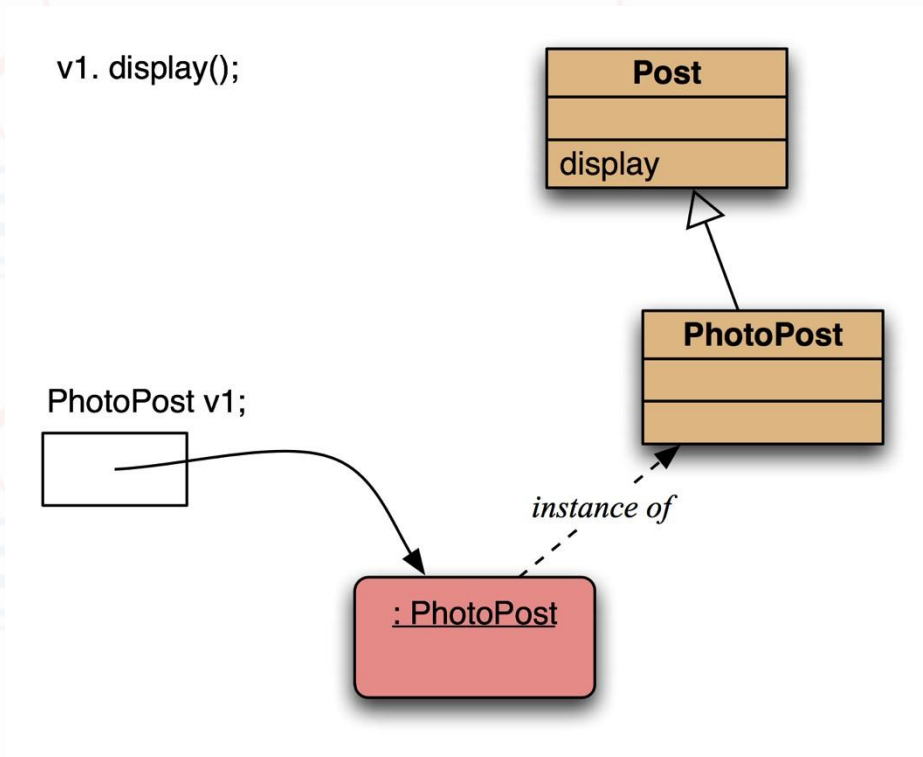
`PhotoPost v1;`



*instance of*

**Žádná dědičnost ani polymorfismus.**  
Je vybrána zřejmá metoda.

# Vyhledávání metody

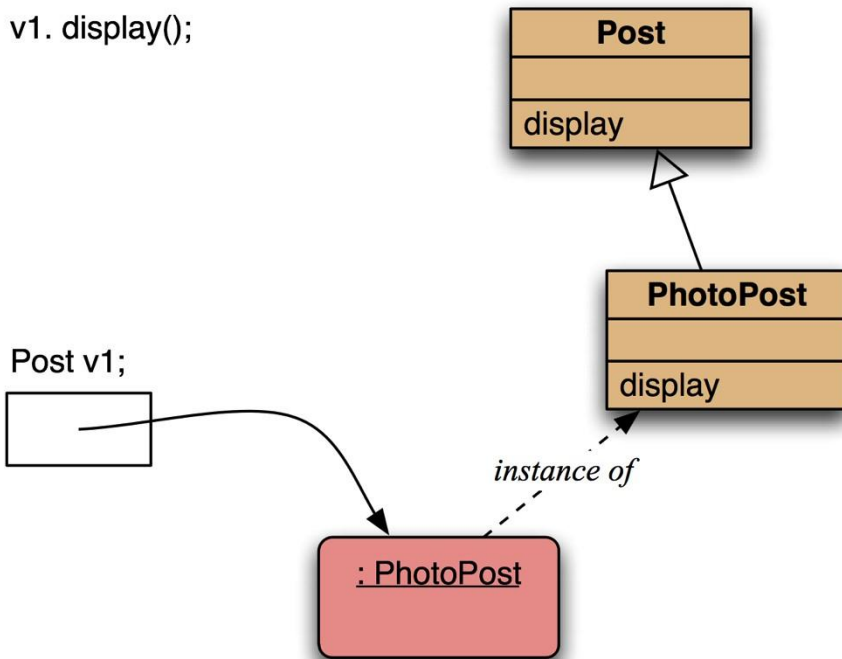


**Dědičnost, ale žádné překrývání.**

Hierarchie tříd je prohledávána směrem „vzhůru“, s cílem nalézt metodu s odpovídající signaturou.

# Vyhledávání metody

`v1.display();`



**Polymorfismus a  
překrývání.**

Hierarchie dědičnosti je  
prohledávána směrem  
„vzhůru“  
a první nalezená verze  
metody je použita.

# Souhrn o vyhledávání metody

- Přistupuje se k proměnné.
- Je nalezen objekt uložený v proměnné.
- Je nalezena třída objektu.
- Tato třída je prohledávána s cílem nalézt metodu s odpovídající signaturou.
- Jestliže není metoda nalezena, je prohledávána nadtřída.
- Toto se opakuje až do okamžiku, kdy metoda je buď nalezena, nebo je hierarchie tříd vyčerpána.
- Překrývající metody mají přednost.

# Volání překryté metody pomocí `super`

- Překryté metody jsou skryté...  
... ale často ještě chceme být schopni je zavolat.
- Překrytá metoda *může* být volána z metody, která jí překrývá.
  - **`super.method(...)`**
  - Porovnejte toto s použitím **`super`** v konstruktorech.

# Volání překryté metody

```
public void display()  
{  
    super.display();  
    System.out.println(" [" +  
                        filename +  
                        "]" );  
    System.out.println(" " + caption);  
}
```

Volání překryté metody nemusí být první v dané metodě.

# Polymorfní metody

- Diskutovali jsme vyhledávání *polymorfních metod*.
- Polymorfní proměnná může uchovávat objekty různých typů.
- Volání metod je polymorfní.
  - Skutečně volaná metoda závisí na dynamickém typu objektu.

# Operátor **instanceof**

- Používá se k určení *dynamickeho typu*.
- Používá se k získání „ztracené“ informace o typu.
- Obvykle předchází přiřazení s přetypováním na dynamický typ.

```
if (post instanceof MessagePost) {  
    MessagePost msg =  
        (MessagePost) post;
```

```
... přístup k metodám třídy MessagePost přes msg ...  
}
```

# Operátor **isInstanceOf**

- Výraz **obj instanceof Type** má hodnotu *true* právě když dynamický typ objektu **obj** je **Type** *nebo podtyp* typu **Type**.
  - Nutno znát **Type** v době překlada.
- Dynamickým ekvivalentem operátoru **instanceof** je metoda **isInstance(Object obj)** ze třídy **Class<T>**.
  - **Class.forName(c).isInstance(obj) ;**
    - je **obj** instancí třídy **c**?

# Metody ze třídy **Object**

- Metody ze třídy **Object** se dědí do všech tříd.
- Každá z nich může být překryta.
- Metoda **toString** je obvykle překrývána:
  - **public String toString()**
  - Vrací znakovou reprezentaci objektu.

# Překrytí metody toString ve třídě Post

```
public String toString()
{
    String text = username + "\n" +
        timeString(timestamp);
    if(likes > 0) {
        text += " - " + likes + " people like this.\n";
    }
    else {
        text += "\n";
    }
    if(comments.isEmpty()) {
        return text + " No comments.\n";
    }
    else {
        return text + " " + comments.size() +
            " comment(s). Click here to view.\n";
    }
}
```

# Překrytí metody **toString**

- Explicitní metody **print** mohou být často ze třídy vynechány:
  - `System.out.println(post.toString());`
- Zavolání metody **println** s objektem má automaticky za následek zavolání metody **toString**.
  - `System.out.println(post);`

# Třída **StringBuilder**

- Použití třídy **StringBuilder** jako alternativy k zřetězení:

```
StringBuilder builder = new StringBuilder();  
builder.append(username);  
builder.append('\n');  
builder.append(timeString(timestamp));  
...  
return builder.toString();
```

# Rovnost objektů

- Co znamená pro dva objekty býti shodný (stejný, totožný)?
  - Rovnost referencí.
  - Rovnost obsahu.
- Porovnejte použití operátoru `==` s metodou **`equals()`** pro řetězce.
  - Řetězce se stejným obsahem mohou být kompilátorem „spojeny“ (viz metoda **`intern()`** ze třídy **`String`**).

# Překrytí metody `equals`

```
public boolean equals(Object obj)
{
    if(this == obj) {
        return true;
    }
    if(!(obj instanceof ThisType)) {
        return false;
    }
    ThisType other = (ThisType) obj;
    ... compare fields of this and other
}
```

# Překrytí metody `equals` ve třídě `Student`

```
public boolean equals(Object obj)
{
    if(this == obj) {
        return true;
    }
    if(!(obj instanceof Student)) {
        return false;
    }
    Student other = (Student) obj;
    return name.equals(other.name) &&
        id.equals(other.id) &&
        credits == other.credits;
}
```

# Překrytí metody hashCode ve třídě Student

```
/**
 * Hashcode technique taken from
 * Effective Java by Joshua Bloch.
 */
public int hashCode()
{
    int result = 17;
    result = 37 * result + name.hashCode();
    result = 37 * result + id.hashCode();
    result = 37 * result + credits;
    return result;
}
```

# Překrytí metody hashCode ve třídě String

```
public int hashCode() {  
    int h = hash;  
    if (h == 0 && value.length > 0) {  
        int off = offset;  
        char val[] = value;  
        int len = count;  
  
        for (int i = 0; i < len; i++) {  
            h = 31*h + val[off++];  
        }  
        hash = h;  
    }  
    return h;  
}
```

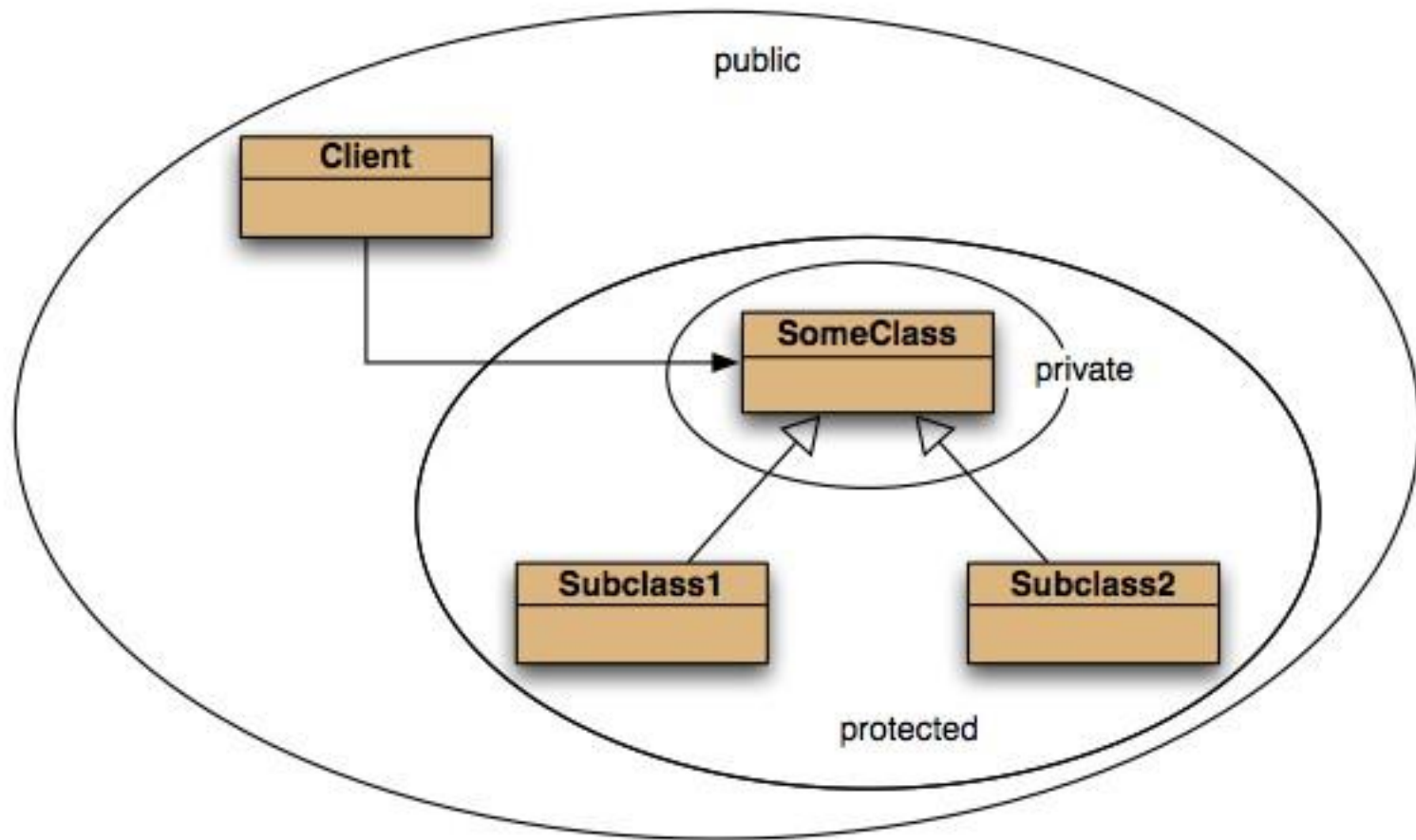
# Překrytí metody `hashCode`

- Kdykoliv je překryta metoda `equals`, je obecně nutné překrýt metodu `hashCode`, aby byl dodržen kontrakt metody `hashCode`.
- Pro objekty, které se rovnají ve smyslu metody `equals`, musí metoda `hashCode` vrátet stejnou hodnotu. (blíže viz [hashCode](#))

# Přístup typu **protected**

- Přístup typu **private** v nadtrídě může být příliš restriktivní pro podtrídu.
- Těsnější vztah daný dědičností je podporován přístupem typu **protected**.
- Přístup typu **protected** je restriktivnější než přístup typu **public**.
- Přesto doporučujeme udržovat instanční proměnné privátní.
  - Deklarujte přístupové a modifikující metody s přístupem **protected**.

# Úrovně přístupu



# Modifikátory přístupu

Tabulka modifikátorů přístupu

Modifier	Class	Package	Subclass	World
<b>public</b>	Y	Y	Y	Y
<b>protected</b>	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
<b>private</b>	Y	N	N	N

# Přehled

- Deklarovaný typ proměnné je její statický typ.
  - Kompilátory provádějí kontrolu podle statických typů.
- Typ objektu je jeho dynamický typ.
  - Dynamické typy jsou využívány při běhu.
- Metody mohou být překryté v podtřídě.
- Vyhledávání metody začíná dle dynamického typu objektu.
- Přístup typu **protected** podporuje dědičnost.