Programozási nyelvek Java

Kozsik Tamás előadása alapján
Készítette: Nagy Krisztián

8. előadás

Öröklődés

- megnyitunk egy osztályt egy másik előtt

  zárt egységeket szeretünk készíteni (láthatósági kérdés: protected)
- A gyakorlatban általában hozzáférés kell a bázisosztály protected adataihoz (mező, metódus,…)
- Veszély: a felüldefiniálással felborítható a bázisosztály eredeti viselkedése (funkciója)
- Gyakran más eszközt választunk, mint az öröklődés

```
class Base{
    public void serve(){…}
    …
}

class Sub extends Base{ }


Sub s = new Sub();
s.serve();  // Lehetnek nem várt következményei!
```

<u>Másképp:</u>

```
class Forwarder{
        private Base b = new Base();
        public void serve(){ b.serve();}
}

Forwarder f = new Forwarder();
f.serve();
```

Ha sok művelet van a Base-ben, mindegyiket beírni bosszantó, a kód is csúnya lesz.
Örökléssel megkapnám mindet.
Base és Forwarder ugyanolyan.

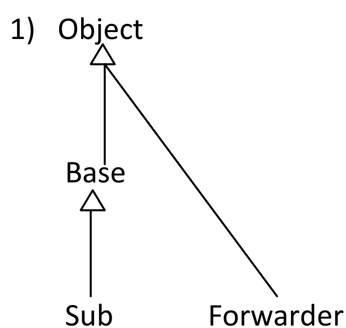
Interface

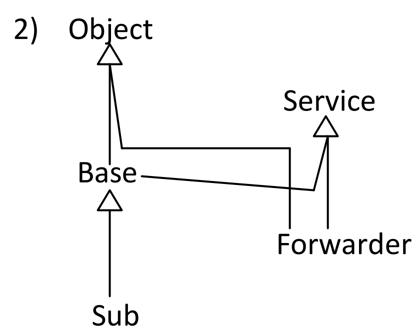
```
interface Service{
	public void serve();
	// public abstract void serve();
}
```

Nem inicializálhatok, nincs konstruktora.
Általában implementáció nélküli műveleteket teszünk bele.

```
class Base implements Service{
	public void serve(){…} // meg kell valósítani az interface működését
}

new Base.serve();

class Sub extends Base (implements Service){
	…
}

new Sub().serve(); // A Sub osztály serve művelete a Service interface-ből jön

class Forwarder implements Service {
	private Base b = new Base();
	public void serve() { b.serve(); }
	…
}

new Forwarder().serve();
```

van közük egymáshoz

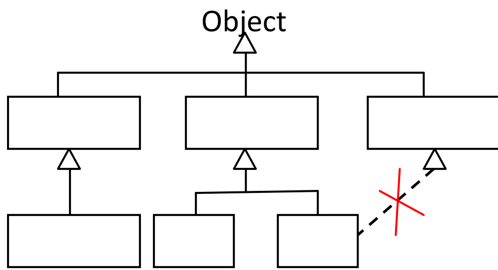


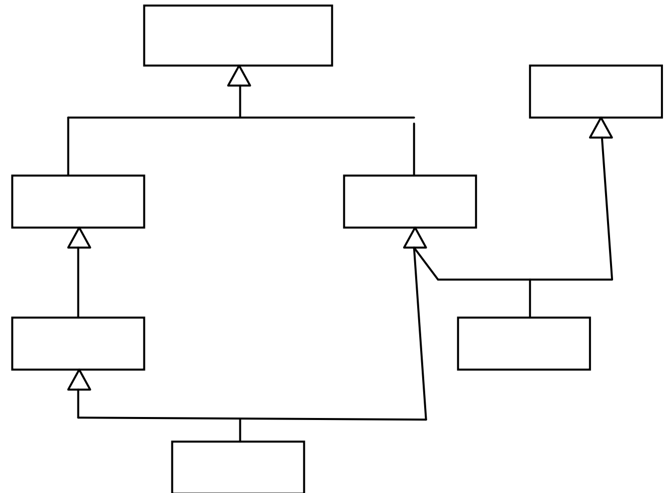


Irányított fa

Irányított körmentes gráf (DAG)
(directed acyclic graph)

class **interface**





extends

irányítatlan körök sincsenek.

parciális rendezés

implements

összeköti a két világot

extends

irányított körök nem, de irányítatlan körök lehetnek

parciális rendezés

Példa:

interface I extends $I_1, I_2, \dots, I_n${

...

}

$I_1, I_2, \dots, I_n$: interface-ek

class: Nem lehet több szülőosztály, de implements lehet több!
interface: lehet több szülőinterface

osztály implements interface ⟶ Lehet több interface (irányítatlan körök)
Az interface egy típus, de objektumot nem hozhatunk belőle létre
Service s; // statikus típus
~~s = new Service();~~ // hibás!
s = new Base(); // dinamikus típus (altípusos polimorfizmus)

$Altípus = extends_{osztály} \cup extends_{interface} \cup implements$ (mint reláció)

Szeretjük úgy használni, hogy interface statikus típus, dinamikus típus pedig egy osztály.

Absztrakt metódus: Olyan metódusok, melyeket nem definiálunk, nincs törzsük, csak deklaráltuk őket.
Az interface-kben csak absztrakt metódusok lehetnek és public static final mezők (ez utóbbi ritka!)

A metódusok public-ok (Nem static, mert nem lenne példányszintű és nem final, mert nem lehetne „felüldefiniálni” (definiálni))

```
interface I extends I_1, I_2, … , I_n{
        public static final int x = 1;
        int y = 2; // automatikusan public static final lesz!
        void serve(); // automatikusan public abstract lesz!
}
```

[public] interface …
- más csomagokban is hozzáférhető lesz
A megvalósító osztály definíciót [törzset] rendel az interfaceben deklarált metódusokhoz.
$\approx felüldefiniálok$ → dinamikus kötés

Megvalósítás szabályai ugyanolyanok, mint a felüldefiniálásé. (később)

Túlterhelés

Metódus: több ugyanolyan nevű metódus lehet osztályon belül, más paraméterlistával.

```
public Base implements Service{
        public void serve() {…}
        // public int serve(){…}          → ez a különbség nem elég a túlterheléshez
        public int serve(int x){          → // ez már igen!
                System.out.print(x);
                serve();    // a paraméter(ek)ből derűl ki melyik hívódik meg
                return x;
}
```

A visszatérési érték típusa nem határozza meg. (hívás után nem kell használnom)

Felüldefiniálás: futás közben dől el, hogy ugyanannak a műveletnek melyik változata fusson.
Túlterhelés: Fordítási időben, nem ugyanannak a metódusnak!

	felüldefiniálás	túlterhelés
döntés	futási időben (dinamikus kötés alapján) a példánymetódus hívását fogadó objektum dinamikus típusa szerint	fordítási időben (statikus típus alapján) az aktuális paramétereknek megfelelően statikus típus alapján
mik közül választhatok	ugyanazon metódus különböző törzsei közül	egy osztály ugyanazon nevű, de különböző metodusai közül

```
class C{
        void m(Base b){ … b.serve(); …} → Sub-beli felüldefiniálás miatt, dinamikus kötés
        void m(Sub s){ … s.serve(); … }

}
```

A fentebbi m() esetén túlterhelésről beszélünk, mivel a formális paraméterekben deklarált típusok eltérnek. (Egyik Base a másik Sub)

```
Base b = new Sub();

new C().m(b); // Túlterhelés m(Base b) –re
```

(HA van void m(Service k), ez is jó lenne, de nem olyan pontos, mint az m(Base b))

class X{
void m(Base b_1, Sub b_2){…}
void m(Sub b_1, Base b_2){…}
}

m(new Sub(), new Sub()) → egyformán pontatlan, fordítási hiba!

NE terheljünk túl altípuson keresztül! (, hogy csak bázis-altípus különbségek vannak)