Objektno orijentirano programiranje

5: Apstraktne klase. Sučelja

Creative Commons

You are free to

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

under the following terms

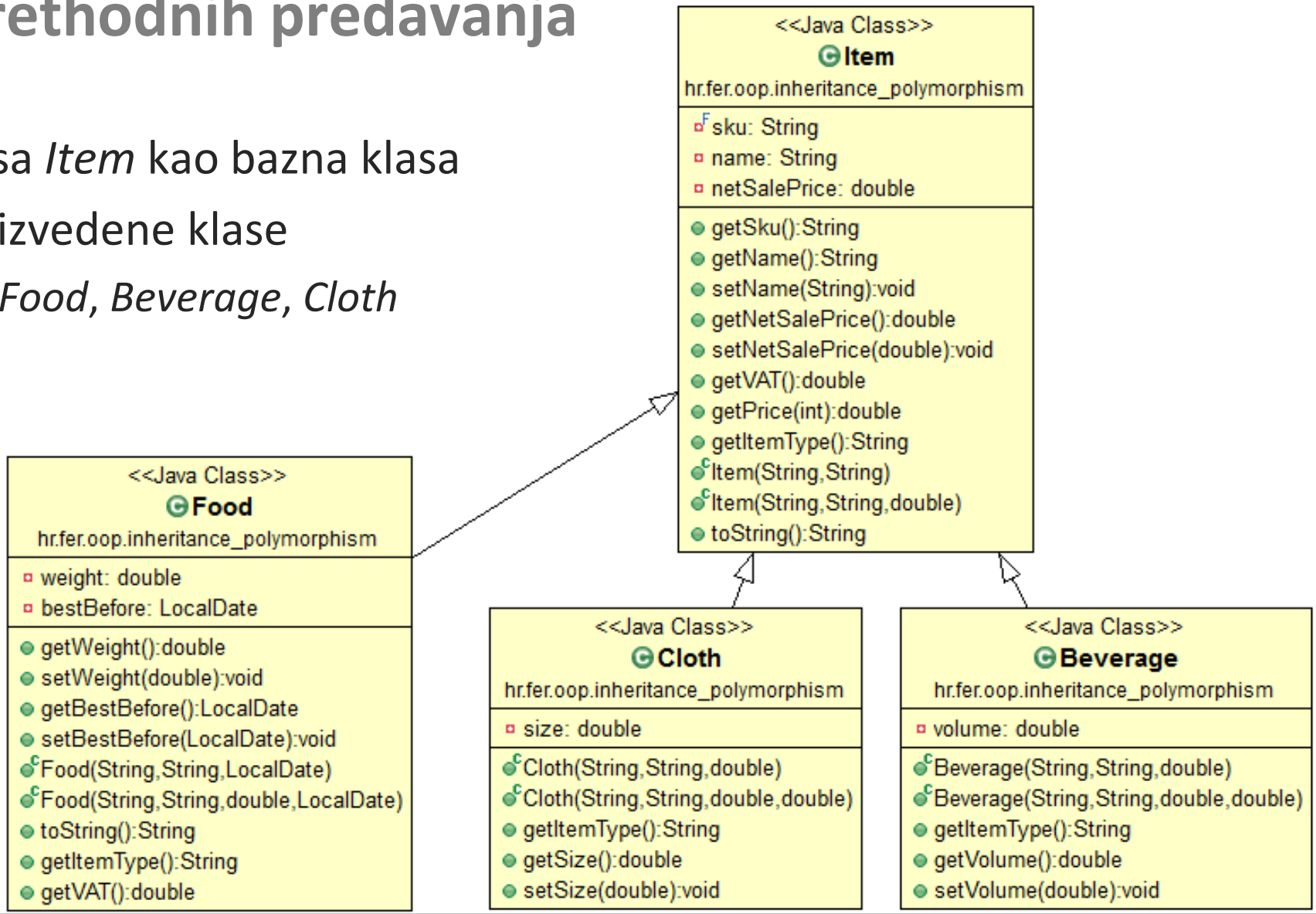
- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

- https://creativecommons.org/licenses/by-nc-sa/4.0/

Podsjetnik na primjer iz prethodnih predavanja

…04_InheritancePolymorphism/…

- klasa *Item* kao bazna klasa
- Tri izvedene klase
  - *Food*, *Beverage*, *Cloth*



(Podsjetnik) Ispis tipa artikla

…04_InheritancePolymorphism/hr/fer/oop/inheritance_polymorphism/Main.java

- Dopustili smo stvaranje prehrambenog artikla kao općenitog artikla, pa je krivo izračunat PDV i nije ispisan tip artikla
  - Na razini artikla ne možemo unaprijed odrediti kategoriju
- Dodatni problem
  - Moguće stvoriti artikl koji nije hrana, piće ili odjeća
    - U primjeru koji slijedi želimo to onemogućiti

```
Item item = new Item("1256", "Domaćica");
item.setNetSalePrice(10);
System.out.format("%s, price: %.2f, type: %s\n",
                        item, item.getPrice(1), item.getItemType());

Food food = new Food("777", "CaoCao", 2.5, LocalDate.of(2016,5,11));
System.out.format("%s, price: %.2f, type: %s\n",
                        food, food.getPrice(1), food.getItemType());
```

```
1256 - Domaćica, price: 12,50, type:
777 - CaoCao, best before: 11.05.2016., price: 2,82, type: Food
```

Apstraktne klase

- Proglasimo klasu *Item* apstraktnom koristeći ključnu riječ *abstract*

  *public **abstract** class Item*

- Primjerak apstraktne klase ne može se direktno stvoriti operatorom *new,* npr. *new Item(…)*
  - ispravno, jer ne smije/ne može nastati artikl „nepoznate” vrste
  - mogu se stvoriti samo objekti tipa *Food*, *Beverage* ili *Cloth*
    - po potrebi možemo definirati nove klase za nove tipove artikala
- Zašto bismo imali apstraktnu klasu? Zato što možemo imati „zajednički nazivnik” za konkretne artikle.
  - apstraktna klasa može imati članove (varijable i metode) koji su zajednički za više vrsta artikala
  - moguće definirati npr. polje artikala ili metodu koja prima artikl
    - (kao u primjeru polimorfizma iz prethodnih predavanja)

Reference apstraktnog tipa

- Iako nije moguće direktno stvoriti objekt apstraktne klase, moguće je definirati referencu apstraktnog tipa…

```
Food food = new Food("777", "CaoCao", 2.5, LocalDate.of(2016,5,11));
Item item = food;
...

item = new Beverage(...
```

- … ili polje tog tipa, gdje su elementi polja reference na objekte tipa koji je izveden iz apstraktne klase (kao u primjeru iz prethodnih predavanja)

```
Item item = new Item[3];
...
item[0] = new Beverage(...
…
item[1] = new Food(...
```

Apstraktne metode

- Kako riješiti problem metode koju ne znamo implementirati, npr. *getItemType* u klasi *Item*.
  - Trenutno se vraća prazni string, ali što bi bilo da je u pitanju metoda koja mora vratiti neku vrijednosti?
    - vratiti 0, -1, neki magični broj?
    - ispisati odgovarajuću poruku?
    - zaustaviti program? …
  - Za razliku od *getItemType*, *getVAT* i *getPrice* imaju smislenu implementaciju koja se može nadjačati, ali ne mora
- Kako natjerati izvedenu klasu da nadjača neku metodu?
  - tako da metoda ne bude implementirana i da se označi s *abstract*

…05_Abstract_Interfaces/…/Item.java

```
public abstract class Item {
        public abstract String getItemType();
        ...
```

Apstraktne klase i apstraktne metode

- Klasa koja ima bar jednu apstraktnu metodu mora biti apstraktna.
- Klasu možemo proglasiti apstraktnom čak i ako nema nijednu apstraktnu metodu.
- Klasa koja naslijedi apstraktnu klasu
  - mora implementirati apstraktne metode iz bazne klase

  ili
  - je i ona apstraktna ako nije implementirala sve apstraktne metode
- Apstraktne metode nemaju kôd
  - Iza naziva metode stavljamo *;* umjesto koda u vitičastim zagradama

```
public abstract String getItemType();
```
- Apstraktne klase mogu imati konstruktore
  - inicijalizira dio koji pripada apstraktnoj klasi i biva pozvan prilikom stvaranja objekta koji ga nasljeđuje.
  - može biti označen s *public*, ali nema svrhe

Pozivanje apstraktne metode

- Može li se apstraktna metoda pozvati koristeći referencu apstraktnog tipa
  - Da! – Apstraktne metode su virtualne metode (polimorfizam)
  - Izgleda kao da zovemo metodu koja ne postoji, ali ta metoda je virtualna te je implementirana u (ne-apstraktnoj) izvedenoj klasi te se poziv konkretna implementacija za konkretni tip objekta

- Može li se apstraktna metoda pozvati iz neke druge metode te apstraktne klase?
  - Da, osim u konstruktorima klasa više u hijerarhiji!
    - Ova opaska za korištenje iz konstruktora vrijedi općenito za virtualne metode i vodi ka nedefiniranom ponašanju, jer klase u stablu nasljeđivanja još nisu inicijalizirane u tom trenutku
      - prisjetiti se redoslijeda izvođenja konstruktora

Primjeri pozivanja apstraktne metode

- Ne-apstraktna metoda klase *Item* poziva apstraktnu metodu klase *Item*
- Što ako je definirano *Item item = …* i pozovemo *item.toString() ?*
  - Ne možemo imati *item = new Item(…* već mora biti nešto nalik
    item = new Cloth(…
  - Klasa *Cloth* ne mora nadjačati *toString*, ali mora implementirati *getItemType*

…05_Abstract_Interfaces/…/Item.java

```
public abstract class Item {
	public abstract String getItemType();
	...
	@Override
	public String toString() {
		return String.format("%s - %s (%s)",
			getSku(), getName(), getItemType());
	}
```

Daljnja specijalizacija klase *Beverage*

- Možemo proširiti pića na različite načine, npr. kao alkoholna pića koja imaju atribut za udio alkohola i nadjačanu metodu za PDV…
- … ili kao mlijeko koje je piće dodatno prošireno postotkom mliječne masti i vrstom mlijeka
  - ograničimo vrste mlijeka koristeći skup pobrojanih vrijednosti

```
public enum MilkType { COW, SHEEP, GOAT, DONKEY }
```

…05_Abstract_Interfaces/…/MilkType.java

  - Napomena: enumeracije su tipovi izvedeni iz *java.lang.Enum*
- Klasa *Milk* označena je s *final* da se spriječi nasljeđivanje
  - želimo zabraniti da netko napravi daljnje specijalizacije mlijeka

```
public final class Milk extends Beverage
        private MilkType type;
        …
```

Zajedničko ponašanje u različitim dijelovima stabla nasljeđivanja

- Neka pića (npr. mlijeko) imaju rok trajanja, ali ne sva pića
- Primijetiti da i hrana ima rok trajanja
- *Ad hoc* ideje
  - Može li se između artikla (klasa *Item*) i hrane (klasa *Food*) dodati još jedna klasa u hijerarhiji koja bi predstavljala kvarljive artikle?
    - Da.
  - Može li mlijeko (klasa *Milk*) naslijediti piće (klasa *Beverage*) i novu klasu za kvarljivu robu?
    - To se zove višestruko nasljeđivanje i odgovor ovisi o programskom jeziku
    - Da u C++-u, ali ne u Javi i C#-u

Višestruko nasljeđivanje

- Scott Meyers, Effective C++

  "Depending on who's doing the talking, multiple inheritance (MI) is either the product of divine inspiration or the manifest work of the devil."

- Inicijalno, izgleda kao da trebamo naslijediti obje klase, ali …
- Trebamo li zaista naslijediti članove obiju klasa, ili nam samo treba definicija zajedničkog ponašanja?
  - nasljeđujemo ako trebamo neku implementaciju (varijable, konstruktore, metode)
    - osim podatka o isteku roka trajanja i *gettera* i *settera* nema drugog zajedničkog koda
  - kvarljiva roba je određena postojanjem ili nepostojanjem metoda *getBestBefore* i *setBestBefore*

Sučelja (1)

- Specifikacija metoda koje klasa mora imati naziva se sučelje i ne modelira se kao klasa (*class*), već kao sučelje (***interface***)
  - metode sučelja nemaju kod (kao apstraktne metode)
    - iznimka su tzv. *default* metode (bit će diskutirano naknadno)
- Nazivi sučelja mogu biti proizvoljni, a u Javi često završavaju s *able*
  - *npr. Iterable, Enumerable, …*
    - ali nije nužno, npr. u slučaju sučelja za rad s kolekcijama nazivi sučelja su imenice *List*, *Set*, ..

Sučelja (2)

- Klase se nasljeđuju, sučelja se implementiraju
- Sučelje može naslijediti neko drugo sučelje
  - proširuje listu metoda koje će neke klasa trebati implementirati
- Klase u Javi mogu naslijediti samo jednu klasu, ali mogu implementirati više sučelja
- Ako bazna klasa implementira neko sučelje, tada i izvedene klase automatski indirektno implementiraju to sučelje
  - Npr. ako *C* nasljeđuje *B* i *B* nasljeđuje *A*, a *A* implementira sučelja *I1* i *I2*, tada referencu na objekt *C* možemo pohraniti u varijable tipa C, *B*, *A*, *I1*, *I2* i *Object*
    - *C* indirektno nasljeđuje *A* (i klasu *Object*) i indirektno implementira I1 i I2
    - C može nadjačati metode iz klase *A* koje implementiraju *I1* i *I2*

Sučelje *Perishable*

- Sučelje *Perishable* definira da sve klase koje implementiraju ovo sučelje moraju imati metode za dohvat i postavljanje roka trajanja.

```
package hr.fer.oop.inheritance_polymorphism;
import java.time.LocalDate;

public interface Perishable {
	public LocalDate getBestBefore();
	public void setBestBefore(LocalDate bestBefore);
}
```

Implementing *Perishable* interface

- Klase *Food* i *Milk* trebaju implementirati metode propisane sučeljem *Perishable*
  - Klasa *Food* već ima tražene metode i treba ih dodati u klasu *Milk*
    - Koristimo oznaku *@Override* kako bismo prevodiocu jasno iskazali našu namjeru da se radi o metodama iz sučelja, a ne nekim drugim istog imena

```
public final class Milk extends Beverage implements Perishable {
	private LocalDate bestBefore;

	@Override
	public LocalDate getBestBefore() {
		return bestBefore;
	}
	@Override
	public void setBestBefore(LocalDate bestBefore) {
		this.bestBefore = bestBefore;
	} ...
```

Operator *instanceof*

- Primjer: ispis samo kvarljive robe iz polja artikala
  - Artikli (tj. objekti) mogu biti tipa *Food*, *Beverage*, *Milk*, *Cloth*, …
- Za provjeru je li nešto kvarljiva roba (tj. implementira li *Perishable*) koristimo operator *instanceof*.
  - *downcast* objekta tipa *Cloth* u *Perishable* bi srušio program

```
private static void printPerishableItems(Item[] items) {
  var formatter = DateTimeFormatter.ofPattern("dd.MM.yyyy.");
  for(Item item:items){
       if (item instanceof Perishable){
            Perishable perishable = (Perishable) item;
            System.out.format("%s, type: %s, use before: %s %n",
                  item, item.getItemType(),
                  perishable.getBestBefore().format(formatter));
       }
  }
}
```

…05_Abstract_Interfaces/…/ShowPerishableItems.java

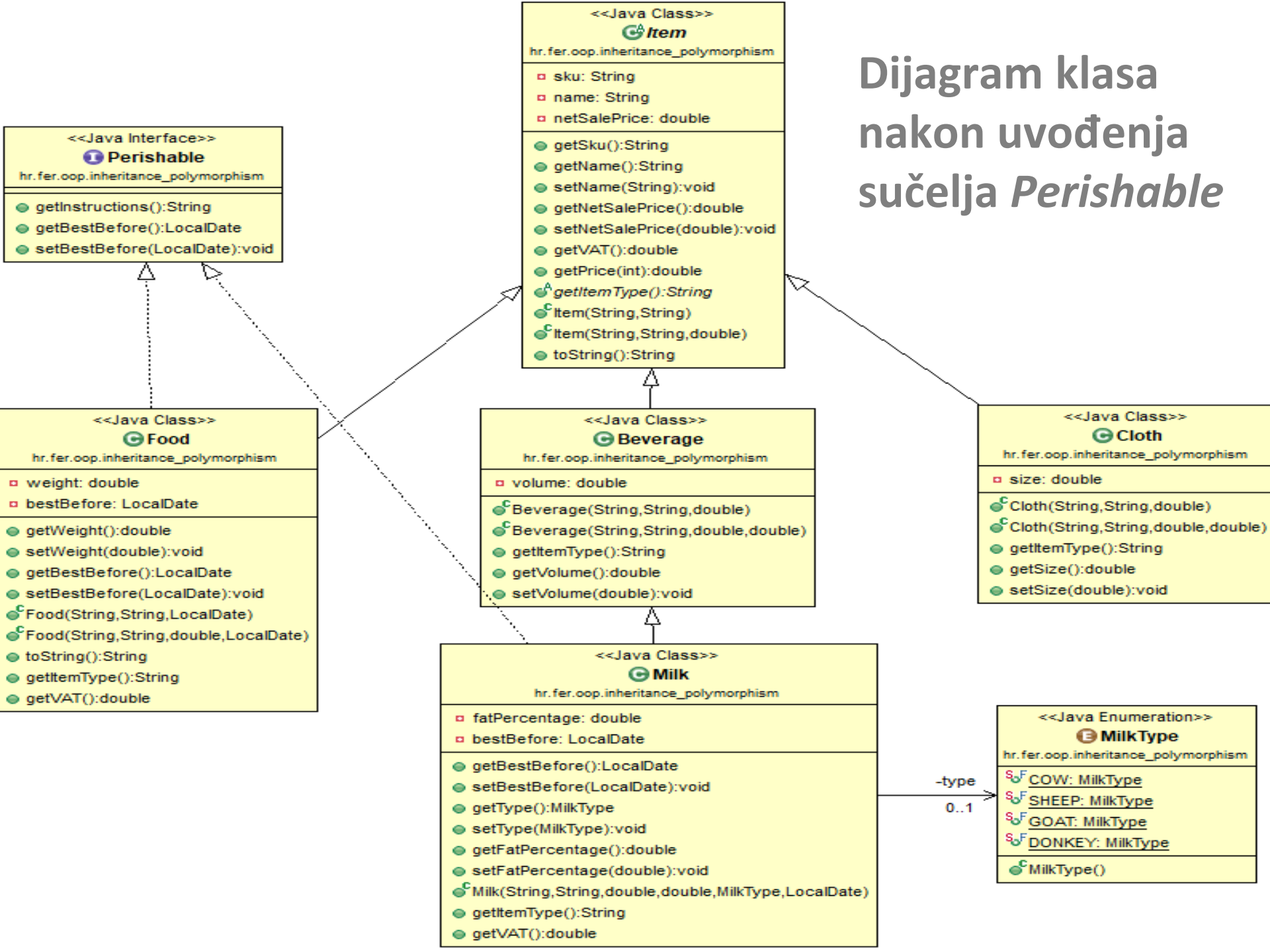
Podrazumijevana implementacija u sučeljima

- Zamislimo da u budućnosti odlučimo proširiti sučelje za kvarljivu robu dodavanjem opisa načina čuvanja artikla
  - sve klase koje implementiraju sučelje *Perishable* morale bi se izmijeniti dodavanjem nove metode
  - alternativno, novo sučelje bi uzrokuje konceptualne promjene koda
- Java ≥ 8 omogućava pisanje podrazumijevanih (engl. *default*) metoda sučelja
  - postojeći kod se ne mora mijenjati, a nove klase mogu nadjačati podrazumijevanu implementaciju

…05_Abstract_Interfaces/…/Perishable.java

```
public interface Perishable {
        default public String getInstructions(){
                return "Keep in dry and cool place";
        }
        public LocalDate getBestBefore();
        public void setBestBefore(LocalDate bestBefore);
}
```

Dijagram klasa nakon uvođenja sučelja *Perishable*



Statičke metode i varijable u sučeljima

- Sučelje ne može propisati da implementacija mora imati određene varijable
  - sučelje je specifikacija (popis) metoda koje implementacija mora imati!
- Sučelje može imati članske varijable.
  - definira se kao obična varijabla, ali smatraju se statičkim i finalnim
- Od Jave 8, sučelje može imati statičku metodu koja ima svoj kod i poziva se nad njim, a ne nad klasom koja ga implementira
  - npr. *InterfaceName.staticMethod(…)*

Sučelja, nasljeđivanje, semantika, JavaDoc

- Semantika sučelja je navedena u njegovoj dokumentaciji
  - prilikom prevođenja, prevodilac ne provjerava poštuje li implementacija semantiku koja je navedena u *JavaDoc-u* sučelja
    - prevodilac samo provjerava sintaksu
- *JavaDoc* se prenosi prilikom nadjačavanja metoda (neovisno radi li se o implementiranju sučelja ili nasljeđivanju klasa).
  - ne treba ga pisati ponovo prilikom nadjačavanja metoda
- Što ako klasa implementira dva sučelja s metodama istog potpisa?
  - U Javi može postojati samo zajednička implementacija tih metoda
  - semantika tih metoda bi trebala biti ista

Potencijalni problemi s podrazumijevanim metodama *

- Korištenje podrazumijevanih metoda u sučeljima može uzrokovati razne probleme. Navedene situacije izlaze iz okvira kolegija, ali ih navodimo radi kompletnosti i za znatiželjne
- Izvor: *Java Complete Reference*
  - Implementirana metoda ima prednost pred podrazumijevanom metodom.
  - Ako klasa implementira dva sučelja s podrazumijevanim metodama istog potpisa, klasa mora imati nadjačanu metodu tog potpisa.
  - Ako sučelje ima podrazumijevanu metodu, a naslijedi sučelje s podrazumijevanom metodom istog potpisa, prednost ima verzija iz izvedenog sučelja.
  - Moguće je eksplicitno pozvati podrazumijevanu metodu iz naslijeđenog sučelja koristeći posebni oblik *super*
    ParentInterfaceName.super.methodName()