Object Oriented Programming in Java

4: Inheritance and Polymorphism

Licence

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.

- The samples and slides are inspired by the Object Oriented Programming course at the University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia.
  - Original materials in Croatian were created by (in alphabetical order): Ivica Botički, Marko Čupić, Mario Kušek, Boris Milašinović, and Krešimir Pripužić under CC-BY-NC-SA licence.
  - Adapted for this course by Boris Milašinović and shared under the same licence
  - https://creativecommons.org/licenses/by-nc-sa/4.0/

Class *Object*

- Java contains class *Object* and an instance of *Object* can be created (although it is not so useful)
- Importance of this class is that enables hierarchical organization of classes with Object as a root of the inheritance tree
- Every reference type (e.g class, but not primitive types) in Java are extended (derived) from *Object* and inherits some elements
  - more about what is inherited later
- Why inheritance?
  - In order to create new class that already have (inherits) some properties (data, behavior, …) that we need and extend it with unique features or behavior

Upcast and *downcast*

- Address of any class instance can be stored in a reference of type Object (***upcast***)
  - With such reference we can call only methods defined in *Object*
- We can try to define more specific reference (e.g. *Point*) from a reference of type Object (***downcast***)
  - It causes "problems" (i.e. exceptions) if we use wrong type
- *Upcast* and *downcast* does not change object on the heap

```
Point p1 = new Point(2, 5);
...                    …04_InheritancePolymorphism/…/swu/oopj/objectmethods/*.java
System.out.println(p1.getX());
Object o1 = p1;  //upcast
//System.out.println(o1.getX()); //compile error
Point po1 = (Point) o1; //downcast
System.out.println(po1.getX());
System.out.println(p1 == po1);
```

```
2.0
2.0
true
```

Comparing and printing objects

- In previous presentation we have used custom methods for printing a point (*print*) and comparing a point to another one (*isEqualTo*)
  - Common need for many classes
- Java already "have" these methods defined in class Object

  boolean equals(Object o)
  - compare an object with another object

  String toString()
  - return objects data as a string (which can be printed later)
- These two methods are inherited, and we have them in our classed although we did not write them explicitly

Using equals and toString from *Object*

- It turns out that using built-in Object methods does not yield desired results
  - *equals* from *Object* compares references and not content
  - *toString* return full class name and some numbers (*hashcode*)

…04_InheritancePolymorphism/…/swu/oopj/objectmethods/*.java

```
package swu.oopj.objectmethods;
public class Main {
        public static void main(String[] args) {
                Point p1 = new Point(2, 5);
                System.out.println(p1.toString());
                Point p2 = new Point(2, 5);
                Point p3 = p1;
                System.out.println(p1.equals(p2));
                System.out.println(p2.equals(p3));
        }
}
```

```
swu.oopj.objectmethods.Point@2f0e140b
false
false
```

Overriding *equals* and *toString*

- If the default behavior of *equals* and *toString* is not appropriate we can write custom implementation (***override*** default behaviors)
  - Annotation *@Override* is not necessary but tells compiler our intention to override those methods
    - Otherwise compiler raises warning due to existence of a method with the same name and arguments in the inheritance tree

…04_InheritancePolymorphism/…/swu/oopj/override/Point.java

```
package swu.oopj.override;
public class Point {
    @Override
    public String toString(){
        return String.format("(%.2f, %.2f)",  x,  y);
    }
    @Override
    public boolean equals(Object obj) {
        Point other = (Point) obj;
        return Math.abs(x-other.x)<1E-8 && Math.abs(y-other.y)<1E-8;
    }
```

Using overridden *equals* and *toString*

- It turns out that using built-in Object methods does not yield desired results
  - *equals* from *Object* compares references and not content
  - *toString* return full class name and some numbers (*hashcode*)

…04_InheritancePolymorphism/…/swu/oopj/override/*.java

```
package swu.oopj.override;
public class Main {
    public static void main(String[] args) {
        Point p1 = new Point(2, 5);
        Point p2 = new Point(2, 5);
        System.out.println("p1.equals(p2) : " + p1.equals(p2));
        p1.setX(1);
        p1.setY(2);
        System.out.println("p1.equals(p2) : " + p1.equals(p2));
        System.out.println(p1);
        System.out.println(p2);
```

```
p1.equals(p2) : true
p1.equals(p2) : false
(1.00, 2.00)
(2.00, 5.00)
```

Override vs Overload

- Override changes the behavior of the inherited method
  - Methods have same signature (name, return type, number and type of arguments)
  - In Java in some cases return type could be different
- Overload adds a new method of the same name but with different number and/or type of arguments

Base and derived class

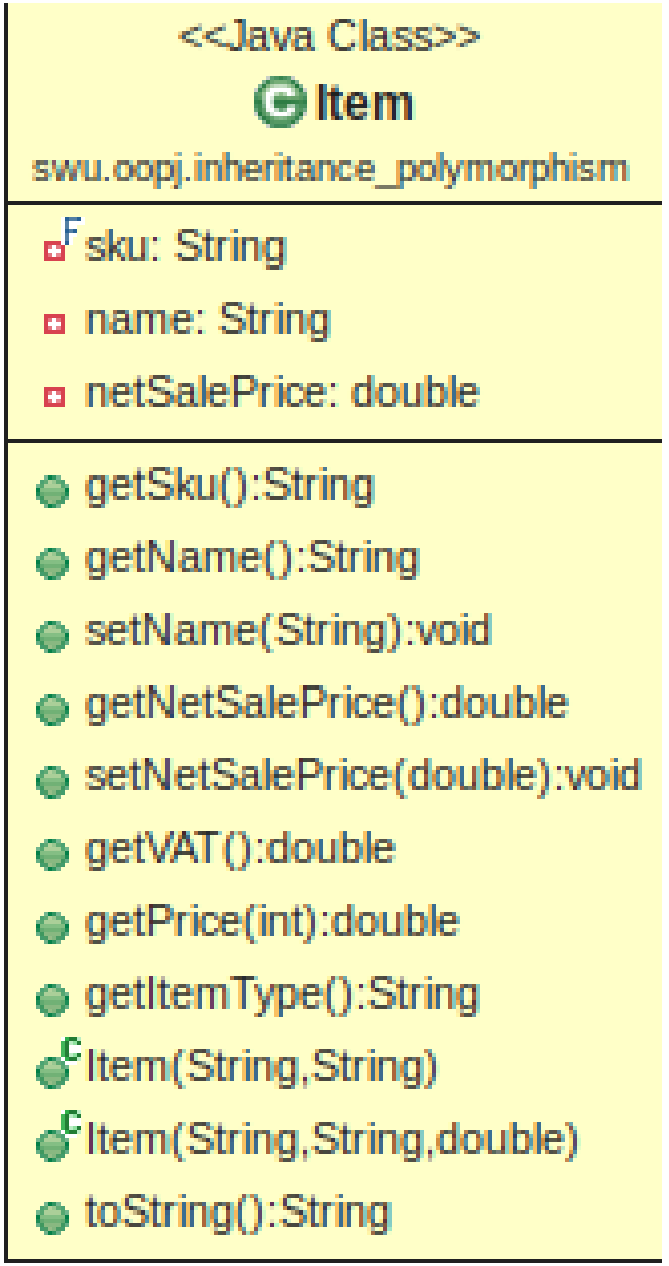
- ***Base class*** or ***superclass*** is a class from which one or more class has been derived
- Class that inherits a class is called ***derived class*** or ***subclass***
- Derived class consist of elements of base class and own elements
  - It inherits members (variables, methods, …) marked as public or protected
    - or without modifier if it is in the same package
    - Constructors are not inherited but can be invoked if the access modifier allows that
  - It cannot access to private members of base class
- Derived class is a *specialization* of the base class
- Base class is a *generalization* of its subclasses

Intro to a complex inheritance example

- Modelling items from an imaginary shop
  - An example is simplified (no database, only few types of items) in order to understand concepts of inheritance and polymorphism
- Every item has
  - Unique identifier (SKU – Stock Keeping Unit)
  - Name
  - Item type (e.g. food, beverage, cloth….)
  - Unit price (net sale price)
  - General value added tax rate (13%)
  - Price for n pieces
    - usually *n * unit price increased for VAT*, but we can model various combinations (e.g. “buy 2 and get one for free”)

Item

- Private attributes (member fields)
- Appropriate getters and setters
  - name and net sale price can be changed
  - sku set in constructor (marked as final)
  - item type and VAT defined at the class level, not for an instance
- *toString* overridden to return string containing sku and name



Item constructors

- Item can be created using sku and name or using sku, name, and price
  - E.g. new Item("1256", "T-shirt") or new Item("1256", "T-shirt", 35.5)
- Constructor chaining

```
package swu.oopj.inheritance_polymorphism;
public class Item {
   ...
   public Item(String sku, String name){
     this(sku, name, 0);
   }
   public Item(String sku, String name, double price){
     this.sku = sku;
     this.name = name;
     this.netSalePrice = price;
   }
  ...
```

…04_InheritancePolymorphism/…/swu/oopj/inheritance_polymorphism/Item.java

Item type, VAT, and overridden *toString*

- *getItemType* returns empty string and getVAT set to 0.13
  - it will be overridden in derived classes for each item type
- Notice that getPrice uses getters for calculation and not variables
  - currently it is the same, but does not have to be because it can be overriden (and it will be shown very soon)

```
public class Item {
  ...          …04_InheritancePolymorphism/…/swu/oopj/inheritance_polymorphism/Item.java
  public double getVAT(){ return 0.13; }
  public double getPrice(int count){
    return count * getNetSalePrice() * ( 1 + getVAT());
  }
  public String getItemType(){ return ""; }
  @Override
  public String toString() {
    return String.format("%s - %s", getSku(), getName());
  }
```

Inheritance examples: food, beverage, cloth

- Food, beverage, and cloth are items (and thus they have sku, name, and price), but every one of them has something unique
  - food (class *Food*) has (e.g.) weight and expiry date
  - beverages (*Beverage*) have volume
  - cloth (*Cloth*) has size
- We want to inherit common properties and features from *Item*
  - e.g. price calculation for n pieces

  and add unique attributes and behaviour
  - e.g food and beverages have different VAT rate that items in general
- Derived classed extends an existing class using *extends* keyword
  - Item already extends Object, but it is by default

```
public class Beverage extends Item {
        ...
}
```

…04_InheritancePolymorphism/…/swu/oopj/inheritance_polymorphism/Beverage.java

Constructors' order of execution

- In order to create *Beverage* instance, part that is extended from *Item* must be initialized.
- Hypothetically, if *Item* had a constructor without arguments, we could write the following code

```
public class Item  {
    public Item() {
       this(…some random sku…, "no name")
       System.out.println("Item constructor");
    }
...
}
public class Beverage extends Item {
    public Beverage() {
        // compiler adds an instruction to call constructor of Item
        System.out.println("Beverage constructor");
    }
```

Ispis:

```
Item constructor
Beverage constructor
```

super keyword to call base class constructor (1)

- Keyword *super* refers to base class members
- *super(zero or more arguments)* calls base class constructor
  - Reminder: *this(arguments)* calls another constructor from the same class
- super or this (if used) must be the first line in a constructor
  - If a constructor does not explicitly invoke a superclass constructor (or use this to do constructor chaining), the Java compiler automatically inserts a call to the no-argument constructor of the superclass.
  - If the super class does not have a no-argument constructor, this would produce compile-time error.
    - As in case of Item that has only constructors with 2 and 3 arguments
  - At the top of hierarchy is *Object* with no-argument constructor

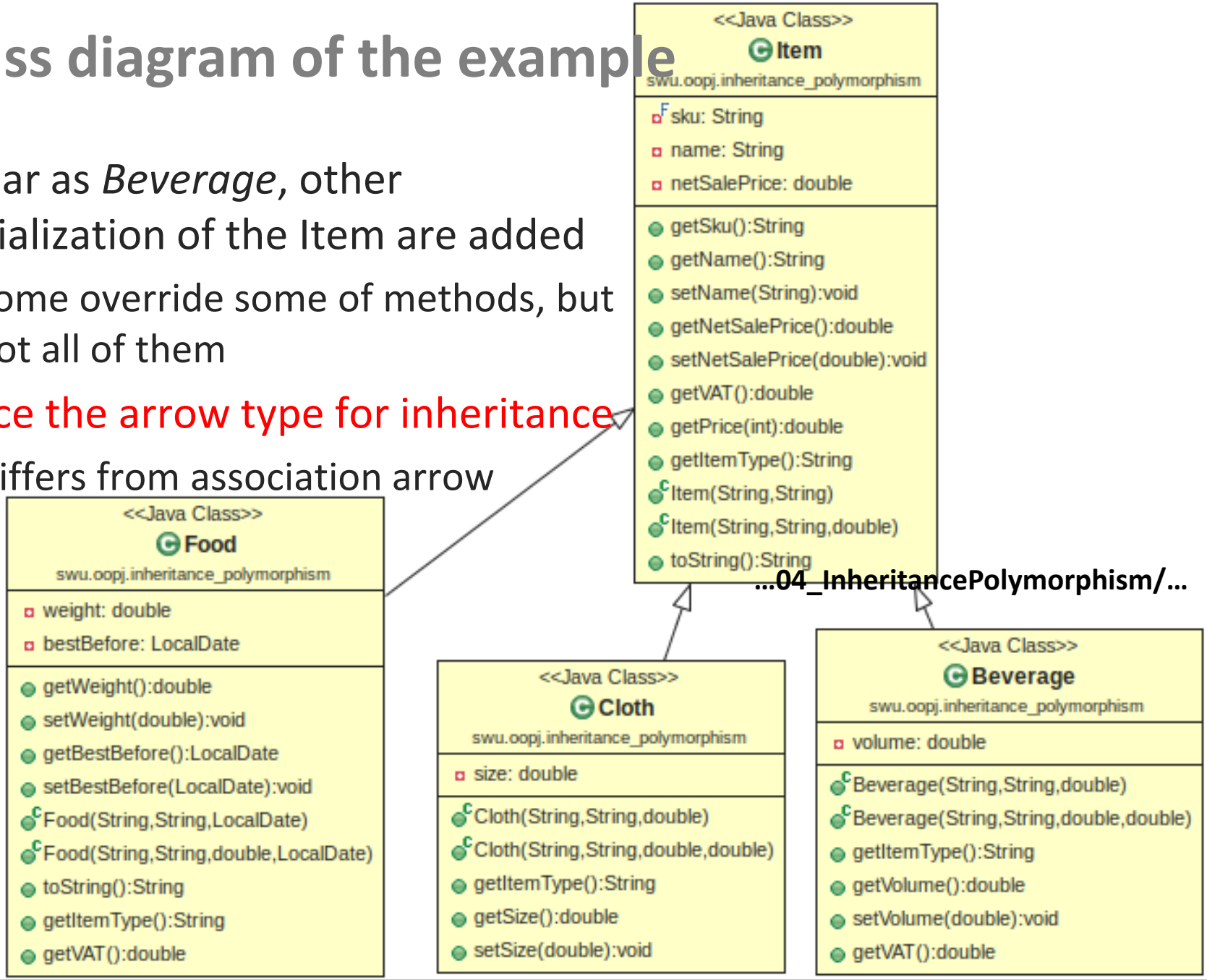
super keyword to call base class constructor (2)

- We can choose which constructor from superclass to call

```
package swu.oopj.inheritance_polymorphism;
public class Beverage extends Item {
  private double volume;
  public Beverage(String sku, String name, double volume){
    super(sku, name);
    this.volume = volume;
  }
  public Beverage(String sku, String name, double price,
                              double volume){
    super(sku, name, price);
    this.volume = volume;
  }
  ...
  @Override
  public String getItemType() {
    return "Beverage";
  }
```

Class diagram of the example

- Similar as *Beverage*, other specialization of the Item are added
  - Some override some of methods, but not all of them
- Notice the arrow type for inheritance
  - Differs from association arrow



Overridden methods

- By deriving from *Item*, classes *Food, Beverage*, and *Cloth* have inherited methods *getPrice, getVAT, getItemType*, …
  - Each of classes writes own version of *getItemType* to return (now known) item type.
  - Beverages and Food have lower VAT rate and thus overrides *getVAT* from *Item*

```
public class Food extends Item {
        ...
        @Override
        public String getItemType() {
                return "Food";
        }
        @Override
        public double getVAT() {
                return 0.6;
        }
```

super for calling superclass method

- Sometimes is useful to reuse overridden method
  - E.g. in for *Food's toString* we just want to extend *Item's toString* with expiry date
  - Does not have to the first line of the method and any (depending of access modifiers) method or variable can be called or used

```
public class Food extends Item {
    ...
    @Override
    public String toString() {
        String s = super.toString();
        DateTimeFormatter formatter =
                DateTimeFormatter.ofPattern("dd.MM.yyyy.");
        s += ", best before: " + bestBefore.format(formatter);
        return s;
    }
```

Note about inheritance, *overriding* and *super*

- In case of inheritance tree with multiple levels, classes on some levels can define own methods, override inherited methods, …
  - but does not have to
- Classes in the inheritance tree inherits the last overridden version of the method, and super refers to inherited methods regardless where in hierarchy it has been defined
  - Cannot use *super.super.method().* This would be both syntax error and breaking the inheritance concept (if it would be allowed)

class A with methods: m1, m2, m3

class B extends A

overrides m1 and defines m4

class C extends B

overrides m2 and defines m5

class D extends C

inherited and overrides m1 (B's version) – does not now and cannot access A's version

inherited m2 (C' version), m3 (from A) m4 (from B), m5 (from C)

An example of use of overridden and inherited methods

…04_InheritancePolymorphism/…/swu/oopj/inheritance_polymorphism/Main.java

- Using reference as an argument in printing runs *toString()*
  - *e.g. "whatever" + food* is equal to *"whatever" + food.toString()*

```
Item item = new Item("1256", "T-shirt");
item.setNetSalePrice(50);
System.out.format("%s, price: %.2f, type: %s%n", item,
                                item.getPrice(1), item.getItemType());
Food food = new Food("777", "Home cookies", 2.5,
                                        LocalDate.of(2020,5, 11));
System.out.format("%s, price: %.2f, type: %s%n", food,
                        food.getPrice(1), food.getItemType());
Beverage beverage = new Beverage("23", "Juice", 10, 2);
System.out.format("%s, price: %.2f, type: %s%n", beverage,
                        beverage.getPrice(1), beverage.getItemType());
```

```
1256 - T-shirt, price: 56.50, type:
777 - Home cookies, best before: 11.05.2020., price: 2.65, type: Food
23 - Juice, price: 10.90, type: Beverage
```

final to prevent inheritance and overriding

- Marking method with keyword *final* prevents its overriding in derived classes
- Marking class with *final* disables further inheritance (that class cannot be extended)
- Compiler will produce error for such attempts

Polymorphism

- Other sciences:
  - Material science: the ability of a solid material to exist in multiple forms or crystal structures known as polymorphs
  - Biology: a condition where one species contains two or more clearly different morphs or forms
- Computer Science
  - *Stroustrup 2007: provision of a single interface to entities of different types*
  - *Cardelli, Wegner 1985: A polymorphic type is a type whose operations can also be applied to values of some other type, or types.*

Polymorphism types

- Ad hoc polymorphism
  - Function and operator overloading
- Parametric polymorphism
  - Templates and generics
- Subtyping
  - Inheritance, overriding and virtual functions

Polymorphism by subtyping

- Base class contains commons methods allowing subclasses to override them
  - Such methods are called *virtual methods*
  - All instance method (i.e. non-static) in Java are virtual methods
- If a methods expect an object of some type, we can call the method providing an object od the derived type
  - E.g. if a method expects an item of type Item, we can call the method with objects of type *Food*, *Beverage* or *Cloth*, because they are also items and derived from class *Item*
- JVM will call virtual method variant (e.g. an overridden one) based on actual type, and not based on reference type
  - *virtual method invocation*
  - *dynamic method dispacth*
  - decision made in runtime, not during compile type

Polymorphism example (1)

- Array of items is array of references to objects of type *Item* or derived from *Item*
  - Beverage, Food and Cloth are *Items*
  - Note: we saw same thing before (*upcast* from Point to Object)

…04_InheritancePolymorphism/…/inheritance_polymorphism/Polymorphism.java

```
public static void main(String[] args) {
        Item[] items = new Item[3];
        items[0] =  new Beverage("23", "Juice", 10, 2);
        items[1] = new Food("777", "Home cookies", 2.5,
                                    LocalDate.of(2020,5, 11));
        items[2] = new Cloth("1256", "T-shirt", 50, 35.5);
        calculatePrice(items);
}

private static void calculatePrice(Item[] items) { ...
```

Polymorphism example (2)

…04_InheritancePolymorphism/…/inheritance_polymorphism/Polymorphism.java

- *getItemType* : virtual method overridden in Food, Beverage, Cloth
- *item* is reference of type *Item*
- Which method to run depends on object type not reference type!
- The same thing for inherited methods *getPrice(int)*
  - This is method is inherited and not overridden, but uses virtual methods getVAT()

```
private static void calculatePrice(Item[] items) {
        double price = 0;
        for(Item item:items){
                System.out.format("%s, price: %.2f, type: %s%n",
                        item, item.getPrice(1), item.getItemType());
                price += item.getPrice(1);
        }
        System.out.println("Total price = " + price);
}
```

```
23 - Juice, price: 10.90, type: Beverage
777 - Home cookies, best before: 11.05.2020., price: 2.65, type: Food
1256 - T-shirt, price: 56.50, type: Cloth
Total price = 70.05
```

Downcast example

- Although the first element in array is a *Beverage* we cannot use methods specific to Beverage if a reference is of superclass type

```
Item[] items = new Item[3];
items[0] =  new Beverage("23", "Juice", 10, 2);
System.out.println(items[0].getVolume()); //compile error
```

- However, we can do downcast back to Beverage

```
System.out.println(((Beverage)items[0]).getVolume());
```

- Compiler will let us also to do this

```
System.out.println(((Beverage)items[2]).getVolume());
```

but this causes program to crash (i.e. causes unhandled exception of type *ClassCastException*) with the message:

```
Cloth cannot be cast to
   swu.oopj.inheritance_polymorphism.Beverage
```

Inheritance, overriding and access modifier

- Overridden method cannot set more restrictive access modifier
  - E.g. in case that *getVAT* was marked as protected in Item, Beverage can have protected and public *getVAT*, but not private

- If the method in superclass is instance method, that the method with the same signature in subclass must also be instance method
- If the method in superclass is static method, that the method with the same signature in subclass must also be static
  - Otherwise there is a compile error

Static methods cannot be overridden

- In case that superclass and subclass have the same method (same name and arguments) subclass hides the method from the superclass
- There is no overriding and dynamic dispatch
  - Decision which method to run is made during program compilation and it is based on reference type

	Instance method in superclass	Static method in superclass
Instance method in subclass	overriding	Compilation error
Static method in subclass	Compilation error	hiding

…04_InheritancePolymorphism/…/hiding_overriding/Main.java