Object Oriented Programming in Java

5: Abstract classes and interfaces

Licence

You are free to

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

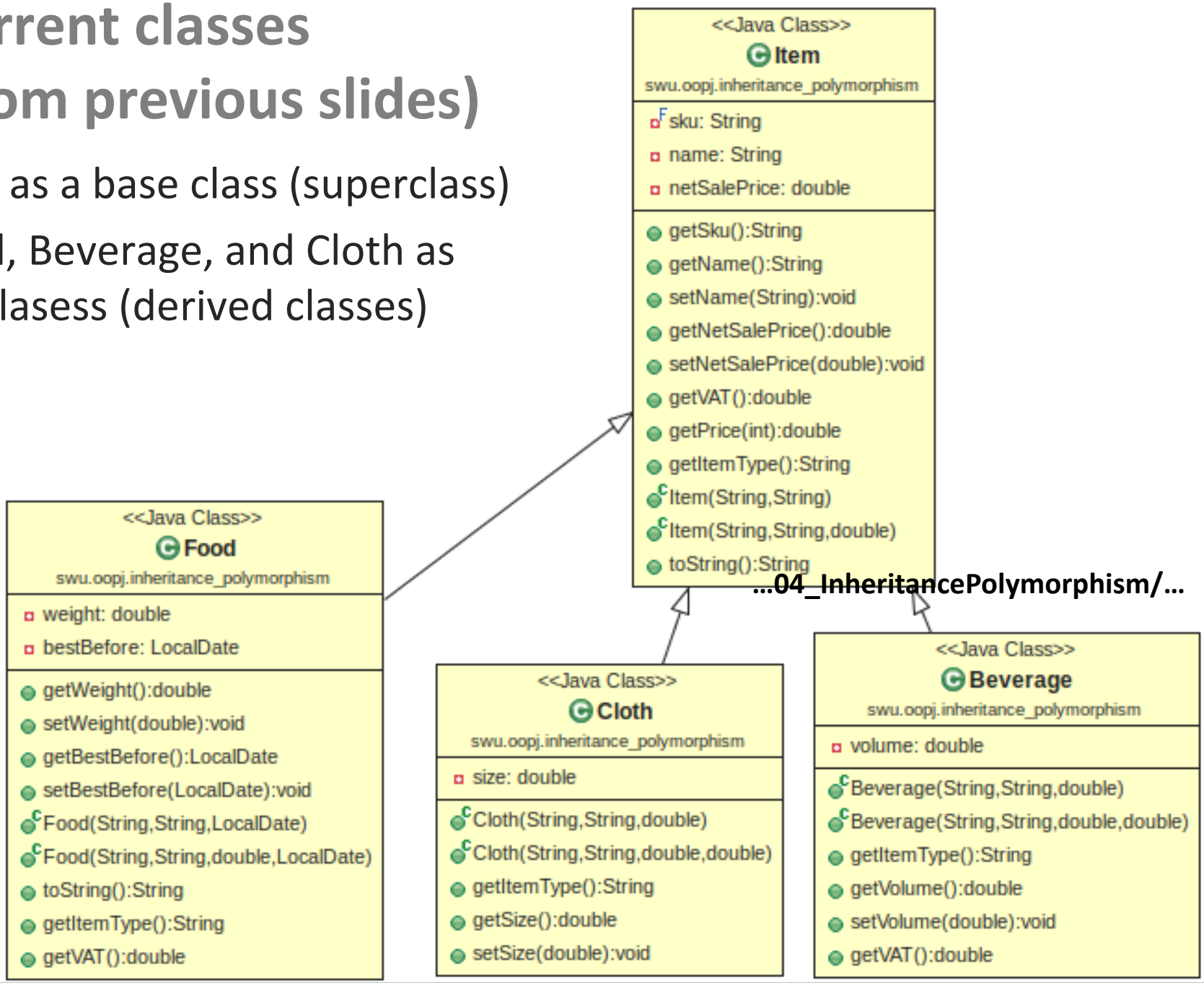
under the following terms

- **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/

Current classes (from previous slides)

- Item as a base class (superclass)
- Food, Beverage, and Cloth as subclasess (derived classes)



(Reminder) Printing item type

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

- In the introductory example we had created clothing item as an object of type *Item* that is not good for several reasons
  - Item type was empty string (we cannot know item type in advance)
  - If *Cloth* override getVAT this leads to incorrect price calculation
  - This also allows creating items that are neither food, beverage or cloth

```
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());
```

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

Abstract classes

- We can prevent creation of objects of type Item and enforce creating object of derived types using keyword *abstract*

  public abstract class Item

- Instance of abstract class cannot be created with *new Item(…)*
  - This is desired behavior as we would like to enforce using only know item types such as food, beverage, and cloth
  - Other unknown types are not allowed
    - we can always define new class for some other item types
- What is the purpose of our abstract Item class
  - It provides common variables and methods of all item types
- We can still use Item as reference, e.g. *Item item = new Food(…*
  - Allows holding different object types in same array as long as their types are derived from *Item*

Abstract methods

- How to implement common methods (in abstract classes) when behavior is not known in advance?
  - Returning empty string looks ok for now, but what happens if we calculate some value (i.e. area of the shape, minimal value, …)
    - Return 0, -1, or some magic number?
    - Write a message?
    - Cause exception and/or crash a program
- Opposite to *getItemType*, *getVAT* and *getPrice* has reasonable implementation that can be overridden but does not have to
- How to enforce derived class to override some methods (i.e. *getItemType*)?
  - By not implementing the method at all and marking it as *abstract*

…05_Abstract_Interfaces/…/Item.java

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

Abstract classes and abstract methods

- A class that contains at least one abstract method must be abstract
- Abstract class does not have to have abstract methods
- A class that is derived from abstract class
  - must implement abstract methods from base class

  or
  - must be also marked as abstract
    - In this case it may implement 0 or more abstract methods
- Abstract classes may have constructors
  - It initialize parts that belong to members of abstract class
  - It may be public, but there is no much use of it, as it can be used from derived classes (using *super* keyword)

Invoking abstract methods (1)

- Can abstract method be invoked if the reference is of abstract type?
  - Yes! - Abstract methods are virtual methods (polymorphism)
- Can abstract method be invoked from another method in the same abstract class?
  - Yes, except in constructors of the classes above in hierarchy tree!
    - This also hold for virtual methods in general and leads to undefined behavior as classes in inheritance tree are still not initialized (remember the order of constructor executing!)
  - It looks we are calling method that does not exist, but that method is virtual, and must be implemented in (non-abstract) derived class

Invoking abstract methods (2)

- *Item's* non abstract method invokes abstract *Item's* method
- What happens if we have *Item item = …* and then *item.toString() ?*
  - We cannot do something like *item = new Item(…)* but we can have item = new Cloth(…
  - Cloth does not have to override *toString*, but must implement (override) *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());
	}
```

Further specialization of *Beverage*

- We can extend beverage in various ways, e.g. alcoholic beverage (with alc% and higher VAT rate), …
- … or as milk that is beverage that defines milk fat percentage and milk type
  - We shall limit possible milk types by using set of enumerated values

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

…05_Abstract_Interfaces/…/MilkType.java

  - Note: enumerations are types that are derived from *java.lang.Enum*
- Additionally we would like to stop further deriving form Milk by making it *final*

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

Common behavior in different part of inheritance tree

- Some beverages (like milk) must be consumed before some date, but it is not the case with all beverages
- Notice that food also must be used before some date
- Ad hoc ideas
  - Can we add a new class into hierarchy between Item and Food for perishable items?
    - Yes!
  - Can Milk extends Beverage and the newly introduces class for perishable items?
    - This would be called multiple inheritance and the answer depends on the programming language!
    - Yes in C++, but not in Java and C#

Multiple inheritance

- 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."

- At the first sight, it looks we need to extends both classes, but …
- Do we really need to inherit both classes, or we just need to define some common behavior?
  - Inherit if you need common implementation (variables, constructors, methods implementation)
  - Perishable items are determined by the existence of expiry date and behavior that that value must be set and retrieved

Interfaces (1)

- Interfaces specifies list of methods that a class must have
  - Those methods does not have a code (like abstract methods)
    - An exception are default methods (discussed later)
- Interfaces are modeled not as classes, but as interfaces (keyword ***interface***)
- Interface names could be various, but often their suffix is *able*
  - *E.g. Iterable, Enumerable, …*
    - but does not have to be, e.g. in case of interfaces that specifies methods for data collections *List*, *Set*, ..

Interfaces (2)

- Classes extends other classes and implements interfaces
- An interface can extend another interface
  - It extends the list of methods that a class that should implement the interface must have
- A class in Java can explicitly extends only one class and implements multiple interfaces
- If the base class of a class implements an interface, then its subclass automatically (indirectly) implement that interface
  - E.g. if class C extends B, and B extends A, and A implements interfaces I1, and I2, then object of type C can be upcasted to reference of type B, A, I1, and I2
    - C indirectly extends A (and Object), and indirectly implements I1, I2
    - C can override methods from A that implements I1 and I2

Introducing *Perishable* interface

- Perishable defines methods for getting and settings expiry date.

```
package swu.oopj.inheritance_polymorphism;
import java.time.LocalDate;

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

Implementing *Perishable* interface

- Classes Food and Milk should implement this interface
  - Food already have them, and we must write them in class Milk
  - We use *@Override* annotation for the same reasons we done that when overriding methods (to tell compiler that we didn't just accidentally used same names as in interfaces that we implement)

```
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*

- An example Primjer: print perishable items from an array of items
  - Items could be Food, Beverage, Milk, Cloth, …
- To check if downcast is possible operator *instanceof*
  - downcasting an Cloth object to Perishable would crash our program

```
private static void printPerishableItems(Item[] items) {
  DateTimeFormatter 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

Default interface methods

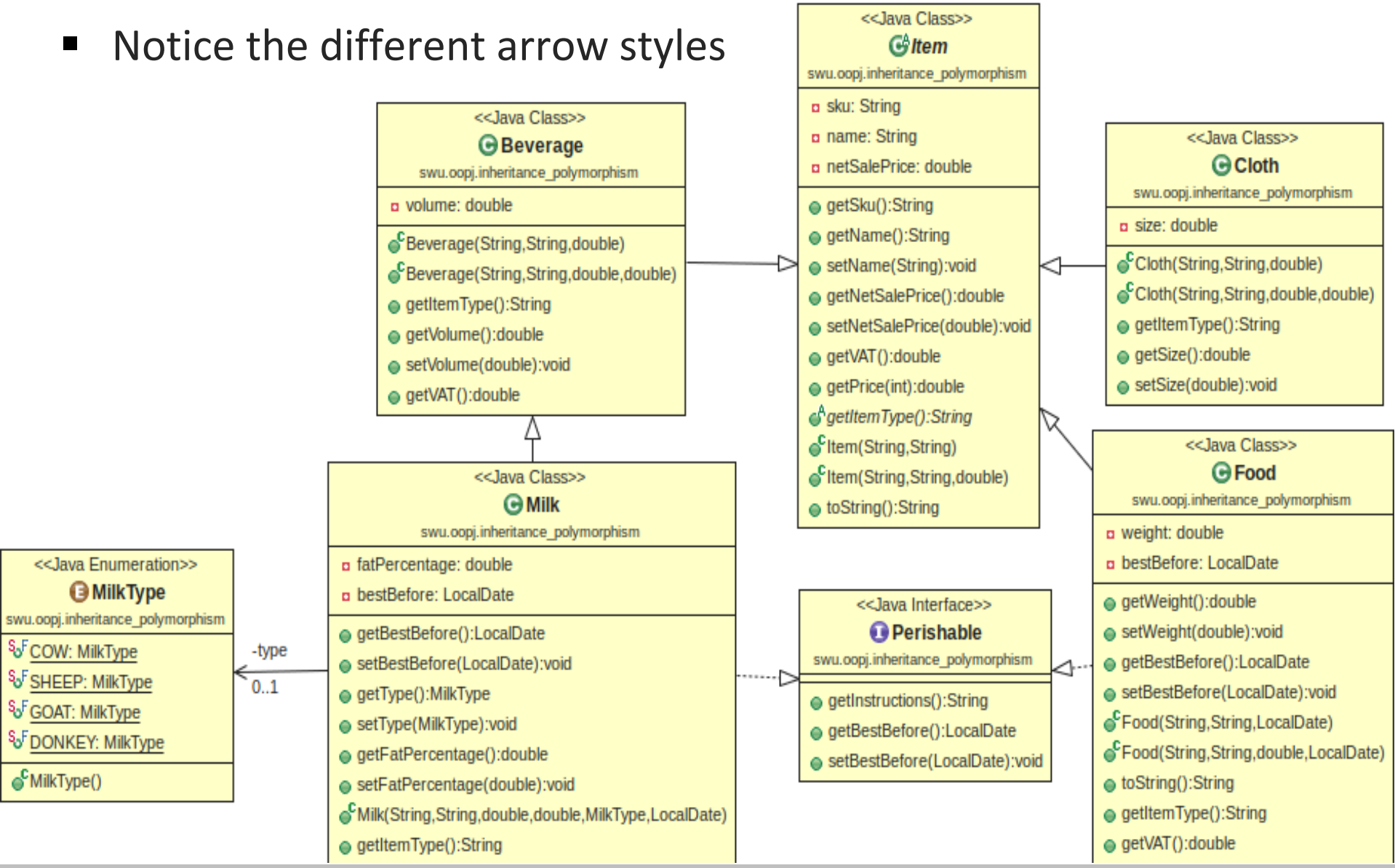
- Suppose we would like to extend *Perishable* interface with a method for keeping introductions
  - All classes that already had implemented *Perishable* must be updated or existing code would not compile
  - Creating new interface forces changes in code
- Java ≥ 8 allows writing default (implementation of such) methods
  - Allows existing code to compile and run successfully
  - Enables classes to override default implementation

…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);
}
```

Class diagram with *Perishable* interface

- Notice the different arrow styles



Static methods and variables in interfaces

- Interface cannot enforce that implementing class must have specific variables or constructors
  - Interface is specifications od methods that class must have
- However, interface can have member variables
  - Defined as “normal” variables, but they are static and final by default
- Since Jave 8, interface can have static methods with appropriate code
  - Like class static methods, run as *InterfaceName.staticMethod(…)*

Interfaces, inheritance, semantics, JavaDoc

- Interface semantics defined by its Java documentation
  - Compiler does check (and it is not able to check) whether the semantics is obeyed
    - Compiler can only check syntax
- *JavaDoc* inherited regardless if it is implementation of interface or class deriving
  - no need to write it again
- What if a class implements two interfaces with the same method?
  - In Java only common implementation of these two methods can be created
  - Thus the methods semantics should be the same
    - otherwise it leads to inconsistency

Some possible problems with default methods*

- Using default methods and implementing interfaces with methods of the same signature can cause “problems”
  - Enumerated for the sake of completeness and curiosity
- Source: Java Complete Reference
  - First, in all cases, a class implementation takes priority over an interface default implementation.
  - In case in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result.
  - In case in which one interface inherits another, with both defining a common default method, the inheriting interface’s version of the method takes precedence.
  - It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of super:
    ParentInterfaceName.super.methodName()