3TC36: Object-Oriented Programming in Java

Class Inheritance (Part 2)

Dominique Blouin
dominique.blouin@telecom-paris.fr

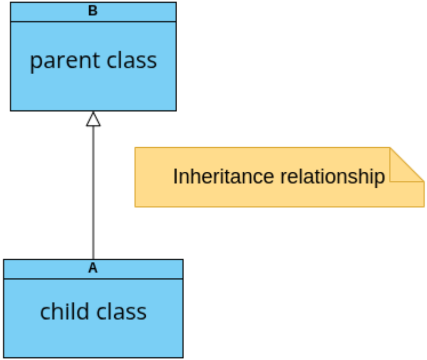
Le February 27, 2026

Learning objectives

- Abstract classes
- Abstract methods
- Best practices for object-oriented modelling

Reminder: Class inheritance

- A class `A` can declare that it **inherits** from another class `B`.
  - Class `A` is called the **child** class or **subclass** of class `B`.
  - Class `B` is called the **parent** class or **superclass** of class `A`.
- Meaning: The child class **inherits** the declarations made in the parent class.



Method redefinition: Example

```java
public class Item {  // An item in a store

    private double netPrice;

    public double getNetPrice() {
        return netPrice;
    }

    public double getVAT() {      // VAT = Value Added Tax
        return 0.185 * getNetPrice(); // 18,5%
    }

    public double getATIPrice() { // ATI = All Taxes Included
        return getNetPrice() + getVAT();
    }
    // ...
}
```

Inheritance: Example

- The `extends` keyword is used to declare the inheritance relationship:

```java
public class LuxuryItem extends Item {

    @Override
    public double getVAT() {
        return 0.33 * getNetPrice(); // 33% tax rate
    }
    // ...
}
```

- `@Override` is an annotation. It serves as an indication to the compiler that the method is being **overridden**. The compiler will verify that this is indeed the case.
- The `@Override` annotation is not mandatory but **highly recommended**.

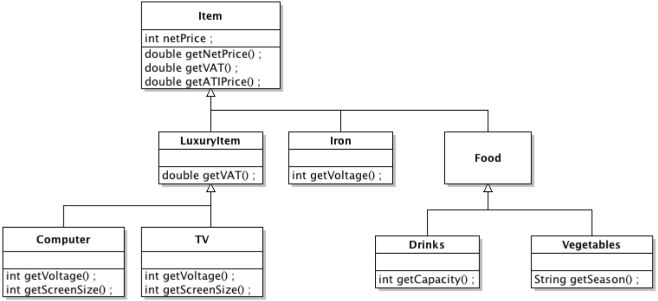
Functions of inheritance

- **Modelling:**
  - Given a class of objects, it can be **partitioned** into subclasses. For example, a class `Shape` can be partitioned into specialized subclasses, such as `Circle`, `Square`, etc.
  - Given a class of objects, it can be **refined** by creating a subclass. For instance, a class `Student` can be specialized into a class `TelecomParisStudent` describing the specific characteristics of Télécom Paris students.
- **Software architecture:**
  - Subclasses of a class **share** the methods and attributes of the parent class.

Meanings of inheritance

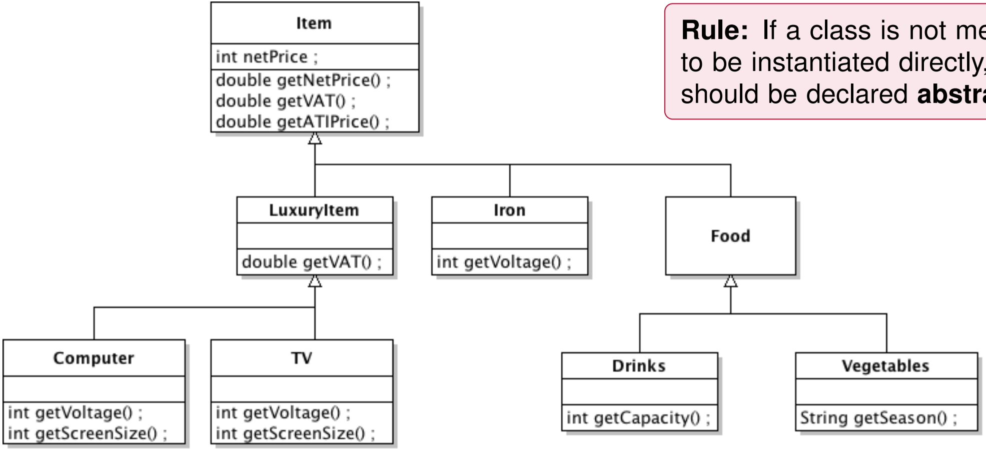
- Is a kind of...
- Is a type of...
- Is a category of...
  - For example, a `TV` is a **kind** of `Item`.
- Is an extension of...
  - For example, a `ColouredPoint` is an **extension** of `Point`.
- Is a specific case of...
- Is a specialization of...
  - For example, a `LuxuryItem` is a **specialization** of `Item`.
- And do not forget the simple **sharing of code.**

Abstract class: Example of store items



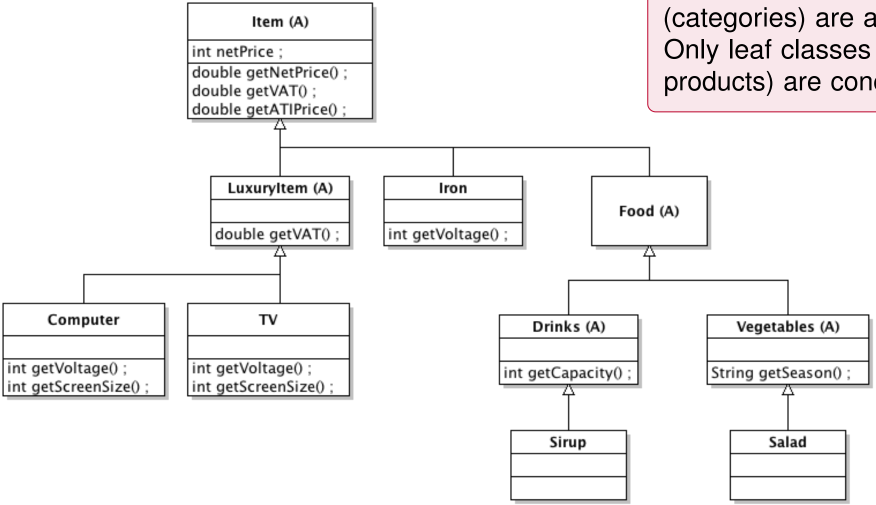
- Note that an object of the class `LuxuryItem` must necessarily be an instance of the class `TV` or the class `Computer`.
  - Creating an instance of the `LuxuryItem` class directly does not make sense.
- This type of class is called an **abstract** class, as opposed to **concrete** classes like `TV` and `Computer`.

Exercise: Which are the abstract classes?



Rule: If a class is not meant to be instantiated directly, it should be declared **abstract**.

Answer: Classes marked with `(A)`



Note: Intermediate classes (categories) are abstract. Only leaf classes (actual products) are concrete.

Example with store item modelling

- To declare that a class is **abstract**, we use the keyword `abstract`:

```java
public abstract class Item {
    // ...
}
```

```java
public abstract class LuxuryItem extends Item {
    // ...
}
```

```java
public class TV extends LuxuryItem {
    // ...
}
```

Example with store item modelling

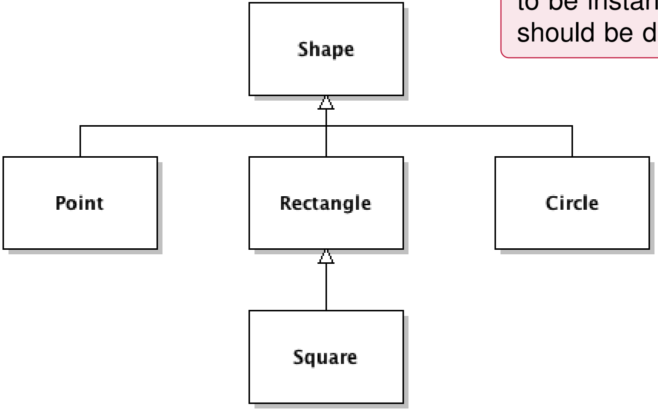
- Declaring a class as `abstract` prevents creating **direct instances** of that class.
- A statement like `new Item()` will cause a compilation error.
- It is still possible to declare variables whose type is an **abstract class**:

```java
LuxuryItem luxuryItem = new Computer(); // This is polymorphism again
```

- Thus, only the methods of the `LuxuryItem` class can be used later, not those of the `Computer` class.

Exercise revisited: A shapes hierarchy

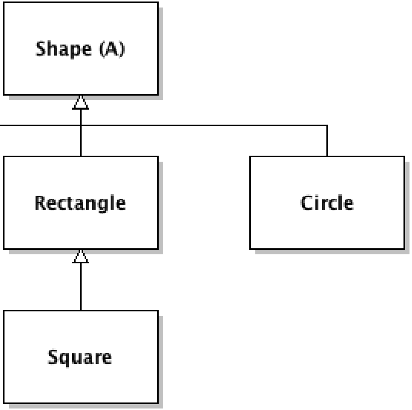
Which are the abstract classes?



Rule: If a class is not meant to be instantiated directly, it should be declared **abstract**.

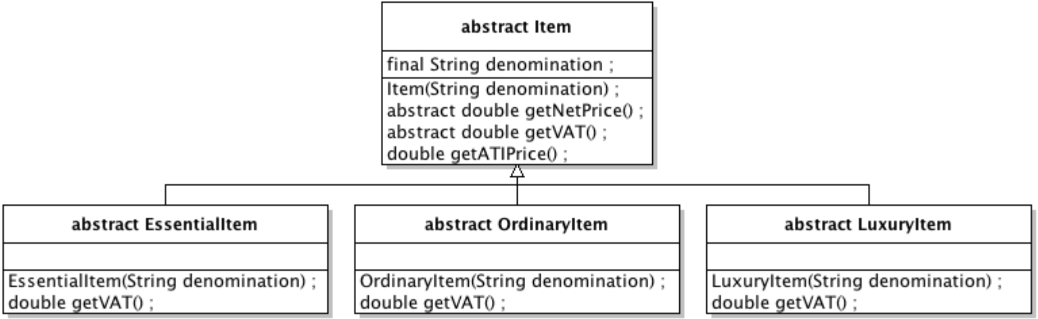
Answer: Classes marked with `(A)`

- The `Shape` class should not be instantiated. Only its subclasses should be instantiated. Therefore, it must be declared as abstract.



Enhanced modelling of store items with abstract classes

- The `Item` class is subclassed three times:
  - The abstract class `EssentialItem` models **essential** items with a **5% VAT**.
  - The abstract class `OrdinaryItem` models **regular** items with an **18.5% VAT**.
  - The abstract class `LuxuryItem` models **luxury** items with a **33% VAT**.



abstract Item
final String denomination ;
Item(String denomination) ;
abstract double getNetPrice() ;
abstract double getVAT() ;
double getATIPrice() ;
abstract EssentialItem
EssentialItem(String denomination) ;
double getVAT() ;
abstract OrdinaryItem
OrdinaryItem(String denomination) ;
double getVAT() ;
abstract LuxuryItem
LuxuryItem(String denomination) ;
double getVAT() ;

Abstract method

```java
public class Item {
    private double netPrice;

    public Item(double netPrice) {
        this.netPrice = netPrice;
    }
    public double getNetPrice() {
        return netPrice;
    }
    // VAT = Value Added Tax
    public double getVAT() {
        return 0.185 * getNetPrice(); // 18,5%
    }
    // ATI: All Taxes Included
    public double getATIPrice() {
        return getNetPrice() + getVAT();
    }
}
```

```java
public abstract class Item {

    private final double netPrice;

    public Item(double netPrice) {
        this.netPrice = netPrice;
    }

    public final double getNetPrice() {
        return netPrice;
    }
    //We don't know the VAT rate, only the subclasses do
    public abstract double getVAT();

    public final double getATIPrice() {
        return getNetPrice() + getVAT();
    }
}
```

- The abstract method `getVAT()` is necessary for the `getATIPrice()` method.
- Since the tax rate is only known by the subclasses, we declare `getVAT()` as **abstract** and **do not provide a method body.**

Revisiting the `EssentialItem` class (1/2)

- The class is abstract for logical reasons: there is no point in creating a direct instance of this class.
- However, the tax rate is known, so the class can provide a concrete method for `getVAT()`.

```java
public abstract class EssentialItem extends Item {

    public EssentialItem(double netPrice) {
        super(netPrice); // Mandatory call to Item constructor
    }

    @Override
    public final double getVAT() {
        return 0.05 * getNetPrice();
    }
}
```

Revisiting the `EssentialItem` class (2/2)

- This method is declared `final` because every subclass must calculate the tax in the **same way**.
- Any concrete class must provide a concrete `getVAT()` method, either in the class itself or in one of its parent classes.

```
public abstract class EssentialItem extends Item {

    public EssentialItem(double netPrice) {
        super(netPrice); // Mandatory call to Item constructor
    }

    @Override
    public final double getVAT() {
        return 0.05 * getNetPrice();
    }
}
```

Revisiting the `OrdinaryItem` class

- The same remarks as for the `EssentialItem` class.

```
public abstract class OrdinaryItem extends Item {

    public OrdinaryItem(double netPrice) {
        super(netPrice);
    }

    @Override
    public final double getVAT() {
        return 0.185 * getNetPrice();
    }
}
```

Revisiting the `LuxuryItem` class

- The same remarks as for the `EssentialItem` class.

```java
public abstract class LuxuryItem extends Item {

    public LuxuryItem(double netPrice) {
        super(netPrice);
    }

    @Override
    public final double getVAT() {
        return 0.33 * getNetPrice();
    }
}
```

Example of a concrete class of `LuxuryItem`

```java
public class Computer extends LuxuryItem {

    private final int voltage;
    private final String model;

    public Computer(double netPrice,
                    int voltage,
                    String model) {
        super(netPrice);

        this.voltage = voltage;
        this.model = model;
    }
    // No getVAT() here. It's inherited as "final" from LuxuryItem
}
```

Modelling of items

- We have just proposed an **improved** model for the classes `Item`, `EssentialItem`, `OrdinaryItem`, and `LuxuryItem`.
- This modelling is perfectly fine. However, a good programmer always worries when seeing the **same instructions** in **different places** in their program.
  - This could indicate a **weakness** in the model.
- The `getVAT()` methods in the three subclasses of the `Item` class are very similar, except for the VAT rate they use.
- Why not store this VAT rate in an attribute of the `Item` class?

An attribute for the VAT rate

```
public abstract class Item {

    private final double netPrice;
    private final double vatRate;

    public Item(double netPrice, double vatRate) {
        this.netPrice = netPrice;
        this.vatRate = vatRate;
    }
    public double getNetPrice() {
        return netPrice;
    }

    public final double getVAT() {  // is now concrete and final
        return vatRate * getNetPrice();
    }

    public final double getATIPrice() {
        return getNetPrice() + getVAT();
    }
}
```

An attribute for the VAT rate

```java
public abstract class EssentialItem extends Item {

    public EssentialItem(double netPrice) {
        super(netPrice, 0.05); // No method overriding, no duplication
    }
}
```

```java
public abstract class OrdinaryItem extends Item {

    public OrdinaryItem(double netPrice) {
        super(netPrice, 0.185);
    }
}
```

```java
public abstract class LuxuryItem extends Item {

   public LuxuryItem(double netPrice) {
      super(netPrice, 0.33);
   }
}
```

Comparison of the two models

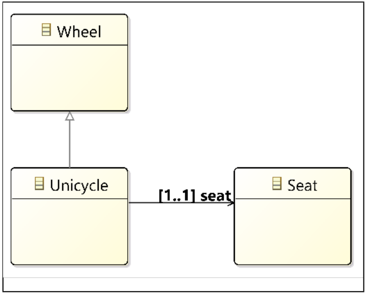
- This new model of the classes `Item`, `EssentialItem`, `OrdinaryItem`, and `LuxuryItem` is neither better nor worse than the previous model.
- However, it avoids the **duplication of code** observed in the three subclasses.
- But, it introduces a certain level of complexity in the root class `Item`.
- Each person is free to prefer one model over the other.

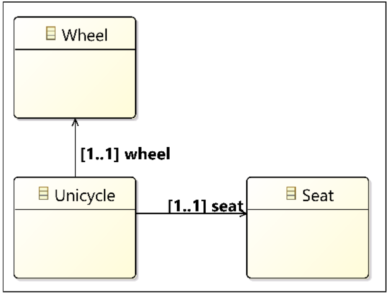
Incrementality and modularity of the modelling

- The modelling of store items can be done *incrementally* and *modularly*:
  - **Incremental**: It is not necessary to develop the entire application for it to function.
    - We can develop and test isolated parts like *televisions* and then move on to another category of items.
  - **Modular**: Each category of item is modelled by a set of classes that are logically independent from others.
    - This could be implemented as a dedicated package.
- This assumes that the base classes and interfaces have been **well designed**:
  - A subsequent modification of the base classes may require reviewing **a large part** of the code (to ensure that everything still works correctly).
  - But, if we choose **not to make needed changes** to the base classes, then the modelling of a new type of item will be poorly programmed.

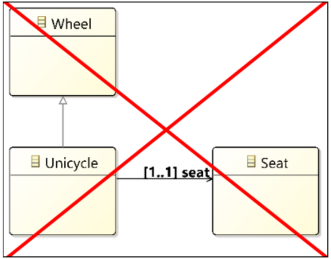
Good use of inheritance

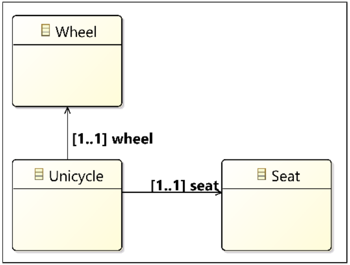
- Inheritance is not always an easy concept to use...
- One must remember the semantics of inheritance:
  - A subclass represents a subset of the objects of the parent class.
- Among the following class diagrams, which is the best model for the unicycle?





Good use of inheritance





- A unicycle **has a** wheel but **is not a** wheel.
- It is important to distinguish
  - **inheritance** ——▷ (`extends` that class) from
  - **composition** ——◆ or ——➤ (a field of that type).

Single responsibility principle

- In object-oriented programming, Robert C. Martin expresses the Single Responsibility Principle as follows:
  - "***A class should have only one reason to change.***"
- A class should have only one responsibility, clearly identified by the class **name**.

Example: Single responsibility or not?

```java
public class Computer extends LuxuryItem {

    private final int voltage;
    private final String brand;

    public Computer(double netPrice, int voltage, String brand) {
        super(netPrice);
        this.voltage = voltage;
        this.brand = brand;
    }

    public void writeToFile(String fileName) {
        // ...
    }
}
```

- Should the `Computer` class be responsible for writing to a file?

Example: Single responsibility or not?

- Saving objects, whether in a file or a database, is **another responsibility**: **data persistence**.
- Therefore, we will use a class **dedicated** to this responsibility.
  - There will be a practical exercise (TP) on this topic ...
- All technical information of the data storage system used will be handled by this class.
- If the storage type changes, only the persistence class will need to change without modifying the data model classes.
- Example:

```
Computer myComputer = new Computer(700,0, 12,0, HP);
EntityPersistenceManager myPersistenceManager = ...;
myPersistenceManager.persist(myComputer); // Data backup
```

Conclusion

- Inheritance is a complex concept with many meanings, and it can sometimes pose conceptual problems (*e.g.*, multiple inheritance).
- Properly modelling the elements of a problem, that is, determining an inheritance tree that is both understandable, logical, and efficient, requires method and experience.
- As often in computer science, there is no absolute method. There are only approaches, best practices, and experience.
- However, we can benefit from the experience of others by consulting some proven best practices.
    - For example, see: https://www.geeksforgeeks.org/best-practices-of-object-oriented-programming-oop/.