Object-Oriented-Programming Basics cheatsheet

Objects

- **Objects** consist of data along with the procedures that operate on that data.
  - We call this data **fields** or **instance variables**.
  - We call the procedures **methods** or **operations** (C++ calls them **member functions**).
  - When we call the method of an object, we say that we **make a request** or **pass a message** to the object.
- **Characteristics of Objects**
  - All objects exhibit both *encapsulation* and *information hiding*.
  - **Encapsulation** is the property that an object's data can only be altered by calling the object's methods.
  - **Information hiding** is the property of protecting internal implementation details from being seen from other parts of the application, for example, by declaring data fields and methods to be **private**.

Types

- Every operation is characterized by a name, the kinds of objects it takes as parameters, and the kind of value it returns. These elements collectively are the **signature** of the operation.
- The collection of all the signatures for the operations that an object can handle is the object's **interface**.
- A collection of signatures is called a **type**. We say that **an object *has* a type** if it has an implementation for each operation specified in the type.
- In Java we can express such types via the formal element called a **Java Interface**.

  > Interfaces specify *what* an object can do, and NOT *how* to do it. The latter is the job of classes.

- Objects can have the same interface/type, but different implementations. A call like `o.draw()` sends the `draw` message to the `o` object. Therefore `o` must implement an interface that contains a `draw` method, but we don't know what specific implementation of `draw` is executed until runtime. At runtime, the implementation of `draw` that the specific object `o` has will be executed. This is known as **dynamic binding**.
- Dynamic binding allows us to write programs based on an object's interface, then swap objects at runtime, as long as they have the same interface. This allows us to vary the implementation without changing the code, so that the result of `o.draw()` can vary depending on which specific method is executed. This ability to substitute

an object with a given interface for another object with the same interface is called **polymorphism**.

Example: We can define two different classes for "points": A `XYPoint` class, which defines a point via its x,y coordinates, and a `PolarPoint` class, which defines a point via polar coordinates, namely the distance `r` from the origin and the angle `theta` that the point forms with the x-axis. *But* both points have a `getX()` method that returns the x coordinate of the point. For the `XYPoint` instances it simply returns the stored `x` value, for the `PolarPoint` instances in computes `r * Math.cos(theta)`. Both classes implement the same `Point` interface. A user who has received an object that implements the Point interface knows that they can do `p.getX()`, but they don't know nor care whether it is the `XYPoint`'s method that gets executed or the `PolarPoint`'s method.

Such a user can for example be given two points `p1` and `p2`, and can compute their x-distance by doing `p1.getX() - p2.getX()`, without needing to know whether the points are both `XYPoint` instances, or both `PolarPoint` instances, or one of each, or some competely different kind of points altogether.

Classes

- A **class** specifies the internal data and method implementations for objects.
- We create objects by **instantiating** a class. We then call the object an **instance** of the class.
- An **abstract class** contains methods that are possibly not implemented, but simply declared to have a specific signature. Abstract classes cannot be instantiated. They can thus be used to express a *type*.
- **Concrete classes** contain implementations of any abstract methods from the abstract classes, and they may also **override** implementations provided by the parent class.

Extension Mechanisms

There are numerous mechanisms that allow us to **extend** the functionality provided by a certain class. The main two mechanisms are the following:

- **class inheritance**, in which we extend the functionality offered by a class by creating a subclass of it. This is often described as an *is-a* relationship.
- **object composition**, in which we use objects of other classes via fields in our new class. This is often described as a *has-a* relationship.

Example. Consider a simple `Grade` class:

```
class Grade {
    private final String letter;

    public Grade(String letter) {
        throwExceptionIfInvalidLetter();
        this.letter = letter;
    }

    public int getPoints() { ... }
    public boolean countsForCredit() { ... }
    public String getLetter() { return letter; }
}
```

We now want to create a UnitGrade class. It needs to allow for both a letter grade and also the amount of units that the class is worth 1, 0.5 etc. We have fundamentally two options for the design of this new class:

1. The *inheritance* approach says that we should create a new *subclass* of the Grade class which also contains a new field, for the units:

   ```
   class UnitGrade extends Grade {
       private final double units;

       public UnitGrade(String letter, double units) {
           super(letter);   // Call Grade constructor
           this.units = units;
       }

       public double getUnits() { return units; }
       // countsForCredit is inherited
       // getLetter is inherited
       public int getPoints() {
           return units * super.getPoints();
       }
   }
   ```

   We use the term extends Grade to indicate that this is a subclass of Grade, and will therefore inherit everything that Grade has. In this instance we automatically get for free the countsForCredit and getLetter methods. We have to modify the implementation of getPoints because it has to take into account the units. We can use the super keyword whenever we have to refer to the superclass.

2. The *composition* approach says that we should create a new class that has a grade field in it and a units field in it.

   ```
   class UnitGrade {
       private Grade grade;
       private double units;

       public UnitGrade(String letter, double units) {
           this.grade = new Grade(letter);
           this.units = units;
       }
       // other possible constructor, receiving an external grade
       public UnitGrade(Grade grade, double units) {
           this.grade = grade;
           this.units = units;
   ```

```
    }

    // Delegations
    public boolean countsForCredit() { return grade.countsForCredit(); }
    public String getLetter() { return grade.getLetter(); }

    public getUnits() {
        return units * grade.getUnits();
    }
}
```

In this case a UnitGrade instance contains a Grade instance as a field, and it can refer to it for information. In particular, some of the methods of UnitGrade simply return a corresponding call to grade. This is called **delegation**.

Let's discuss advantages and disadvantages of the two approaches:

- Inheritance is a bit easier to understand, and requires less code to implement. Composition on the other hand requires more work, via delegation for example.
- Inheritance fixes the implementation of Grade at compile-time. If we wanted to instead use a subclass of the Grade class (maybe some specialized "pass/fail grade subclass", we cannot do that. On the other hand with composition, the field grade can belong to any subclass of Grade. *Composition allows the specific class that is used to be determined at runtime, and to even change in the lifetime of an application.*

Essentially, *inheritance is a static compile-time source-code dependency, while composition is a dynamic run-time dependency.*

In UML notation, these two dependencies are drawn differently: inheritance is a hollow-point arrow, often drawn in a consistent vertical direction, while composition is a filled-point arrow, often drawn in a horizontal direction.

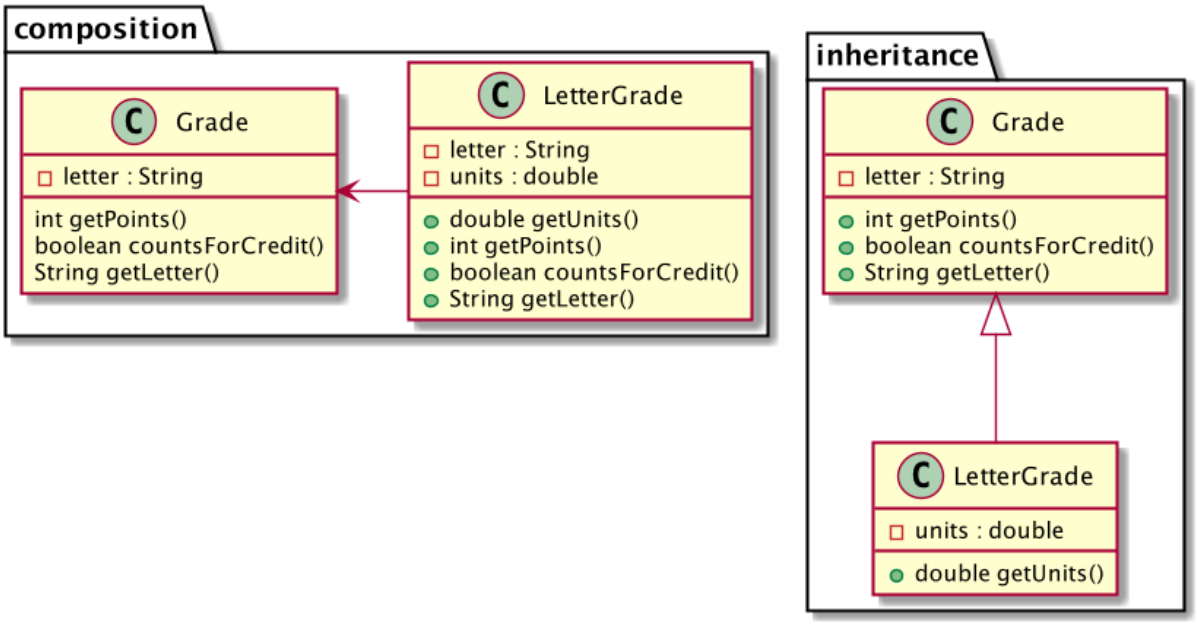

Figure 1: Inheritance vs Composition