Object-Oriented-Programming Basics

Code references are to: OOP Example code: Numbers[1]

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.
  - For example in the ExpressionTest class[2] in the test method sumExpressions, there are three objects named one, two and sum, and we send the "message" getValue to the object sum when we write sum.getValue().
- **Characteristics of Objects**
  - **Encapsulation** is the property that an object's data can only be altered by calling the object's methods. This allows us to control essential internal properties of the object, and can be certain that noone can mess with those properties. You can think of objects a bit like biological cells, where the inner parts are safely protected against outside influence, and the only way to effect a change is by interacting via specific processes on the cell's membrane.
  - **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**. Someone looking at a cell can't really tell what the cell is made of.
    - A good example of this is the SumExpression class[3]. It holds in it two other subexpressions, but the details of how it does so are completely hidden from the outside world: They could be in two separate instance variables, they could be in an array, a list etc. Noone is the wiser and the rest of our application doesn't care; that's the SumExpression's business.
  - **Message passing** is how objects communicate with each other. They pass messages to each other telling them to do something. This is an essential element of object-oriented-programming:

    > In object-oriented method calls / message passing, the sender *sends a message to a receiver*, and the two sides are very loosely coupled to each other. The receiver does not know who sent them the message, and the sender has no idea *how* the receiver will handle that message.

[1] https://github.com/sdp-resources/expressions
[2] https://github.com/sdp-resources/expressions/blob/master/test/ExpressionTest.java
[3] https://github.com/sdp-resources/expressions/blob/master/src/SumExpression.java

As an example of this, consider the getValue implementation of the SumExpression class[4]:

```
public int getValue() {
    return term1.getValue() + term2.getValue();
}
```

Here term1 and term2 are Expression objects, and we call their getValue method. These could be IntegerExpression objects or SumExpression objects; we don't know, and we don't care. We are simply asking them to response to the getValue message. Objects of different classes will respond in different ways.

Types

- Every operation is characterized by a name, the kinds of objects it takes as arguments, and the kind of value it returns. These elements collectively are the **signature** of the operation. As an example, the getValue method takes no arguments, and returns an integer.
- 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**, also some times referred to as **late 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

[4] https://github.com/sdp-resources/expressions/blob/master/src/SumExpression.java

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.
- In Java, we have a distinction between *abstract classes* and *java interfaces*:
  - Both abstract classes and java interfaces may contain declarations of methods but not provide implementations.
  - Abstract classes may have their own instance variables, java interfaces cannot.
  - Abstract classes may have implementations for certain methods. Java interfaces typically do not (although the default keyword does allow these methods, you should be hesitant in their use).
  - A class can only *extend* from one abstract class, but it may *implement* many java interfaces.