A Guided Introduction to Thinking *With* and *About* Object Oriented Programming

1 Introduction to Abstraction

The purpose of learning computer science, especially at the high school level, is not to memorize syntax. Rather, we are working to develop approaches to solving problems. Some of the best tools that we have to solve problems are programming languages.

Almost all programming languages help solve problems. Some languages are better suited to specific problems than other languages. There are many elements of a programming language that help solve problems. These elements can include built-in commands, data types, as well as the structure of the language itself.

When beginning a programming language, it is common to struggle over syntactical features such as commas, semi-colons, or indentation. Although, those elements must be correct for programs to work - they are *minor details* that simply support the true purpose of a programming language - **solving problems**. The structure of the programming language itself determines how we solve problems - namely, how we *de-construct* and think about problems.

When discussing programming languages, it is common to refer to *low level* and *high level* languages. By level, we are not refering to complexity but rather to the level of ***abstraction***. Whereby ***abstraction*** refers to a general class - a broader collection of characteristcics or qualities. For example, when discussing the color *red*, red does not refer to one specific color of one specific object, but rather refers to a color characteristic shared across objects that English speakers refer to as red.

With programming languages, *abstraction* can be seen in the amount of specificity commonly required and utilized across a program. For example, does each instance of *red* require a specific hex combination for each pixel on a screen to be manually specified or can the programmer more generally, simply refer to *red*?

A *low level* programming language such as **C** might require the programmer to be very specific and provide low level specifics such as actual memory bits of the computer. By contrast, a *high level* programming language such as **Processing**, **Python**, or **Java** may very well hide such details from the programmer - allowing the programmer to focus on more *abstract* i.e. 'big picture' ideas.

2 Object Oriented Programming

What is Object Oriented Programming (OOP)? Quite simply stated, OOP is programming structured around *objects* - conglomerations of qualities and characteristics - rather than always detailing specific interactions with the computer's memory. This *object oriented* approach is intended to simplify work for the programmer and improve programming efficiency by closer approximating real world complexity and interactions.

2.1 Generalization

A programmer can work to defining interactions across entire *classes* of objects rather than having to specify interactions for each specific instance. ***But what does that mean????***

Imagine - if every time someone bought a new pair of shoes they had to learn a specific system for tying the laces on that particular pair of shoes. That would be highly inefficient. Rather, we as a society have developed a few methods of tying shoes. Learning to tie one type of shoe readily transfers to tying other types of shoe laces. The laborious process of learning to tie a specific type of shoe lace does not have to be repeated each time. Similarly, imagine that a shoemaker would not want to invent and create a new system for tying shoes each time they made a pair of shoes. Rather, a system would be created for tying shoes and that system could be *inherited* by each pair of shoes that that shoemaker or other shoemakers could use.

2.2 Classes

Imagine further - someone has developed a good underlying design (or *template*) for shoes. In OOP, this can be refered to as a ***class***. In essence, a *class* is like a recipe that can be used as the basic for making many similar, but differentiated, *objects*. The shoemaker could create a *class* (template, recipe, etc.) for a generic shoe design. That class could then be used to mass produce however many shoes of that type (with whatever specified variations) that the shoemaker wants. In terms of OOP, the shoe template would be the *class* used to create individual shoe *objects*, whereby each pair of shoes would be considered an *instance* of **Class** Shoe.

As programmers, and problem solvers (e.g. think about Algebra), the goal is not to spend all our time focused on very specific problems. Rather, we attempt to represent and solve as many problems as possible at once. Sometimes, it may indeed be necessary to specify the behaviors for very particular interactions in very particular contexts. Cf., we may use particular greetings in particular situations (e.g. a specific handshake for a specific friend)- but before we develop specific greetings we develop a generalized greeting such as *Hi!* that can be broadly applicable.

But what does that mean???? In short, as programmers and people, it is not time efficient to develop specific greetings for each individual person

and take that approach to developing greetings for each possible random encounter. Someone could spend the better part of a month just planning out what they might say during a quick trip to the grocery store. Much more time effective is to plan a *generalized* i.e. *absracted* approach and simply plan on saying "hello" to everyone and plan exceptions from there. For example, plan on saying "hello" to everyone except for people whose name is known - then say "hello `$personsname`." This *abstracted* approach is the essence of OOP.

In Java, or other similarly *object oriented* programming languages, a class might simply be defined by something such as:

```
Class Person{ }
```

However, such a code by itself is useless - because this `Class Person` does not actually attribute any qualities to a person nor does it provide a way to create more Person objects i.e. people.

```
int main() {
  printf('hello world'):
  return 0;
}
```