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.

2.3 Data types

One of the biggest, most important concepts in programming, and perhaps in the modern world, is that there are different types of data. We use different types pf data to answer different types of questions. For example, if someone asks "What type of cookies are those?" - the response "12" is not helpful or useful. Similarly, if someone asks "How many cookies do you have?" and is answered with "chocolate chip" - the response is neither helpful nor useful. A specific question requires a specific type of response.

In order to solve any problem, it is first important to understand what the problem is, the characteristics of the problem, and the likely characteristics of the solution. For example, if we are deciding on the types of cookies for a class party - some of the information we might need are numbers and type of cookies.*Number* and *Type* would be the data types we would use to discuss cookies. More cookie specific data types might include *sugar-type* or *leavening-agent*. As can be seen with the cookie example - certain data types make more sense based on the environment. In a cookie specific environment *sugar-type* and *leavening-agent* are very useful. However, in the design of cars they would basically be useless.

Data types in programming are those types of data that have been found to be most useful in programming i.e. computer related tasks. Therefore, each programming language has its own built in set of a data types. There is significant overlap - because there is significant overlap in the function of purpose of computer languages - also there is significant overlap because all computer languages must be designed to work on computers i.e. the actual specifics and architecture of processors determines the shape that a programming language takes - e.g. certain data types reflect the amount of space available in the computer's working memory.

2.3.1 Data Types in Java

In Java, the built-in i.e. hard-coded data types are referred to as *primitives*. The *primitive* data types in Java include:

- **Whole numbers**
  - **byte - a whole number (*integer*) between -128 and 127 (8 bits)**
  - **short - an *integer* between -32768 and 32767 (16 bits)**
  - **int - an *integer* between -2,147,483,648 and 2,147,483,647 (32 bits)**
  - **long - an *integer* between 9,223,372,036,854,775,808 and 9,223,372,036,854,755,807 (64 bits)**
- **Floating points (decimal numbers)**
  - **float - a decimal (*floating point*) (32 bits)**
  - **double - a decimal (*floating point*) (64 bits)**
- **Non numerical**
  - **char - a Unicode character e.g. one single letter, one single number, one single type of punctuation, etc.**
  - **boolean - true or false**

See also the official Java documentation.

2.3.2 Strings

Another very common data type, that is **not** a *primitive* is the `String`. A `String` is simply an *array* (i.e. list) of characters. For example, `String name=''John Doe''` contains all the characters and spaces for J-o-h-n-space-D-o-e. Because *Strings* are *objects*, they are able to perform actions (*methods*) not readily available to other primitives.

2.4 Using Data Types

In short, *classes* are recipes that create *objects* which do things (with *methods*) and contain *data*. There is perhaps no absolutely perfect way to organize data. However, some approaches to working with data are more useful than others. At the end of the day, programming, and indeed much of what we do in our lives, revolves around the organization and restructuring of data in new and useful ways.

Java is an *object oriented* programming language - which means that the language is structured to work with data as aspects of *objects*.

Earlier, the example of the empty `Class Person` was given, which was basically an empty recipe with no way of being made and no associated data. In order to do anything meaningful however, we will want to create *objects* that have data and can do things i.e. have *methods*. [1]

2.5 First Worked Example - An *object*-first approach

In Java, basically everything occurs in relationship to some *Class*. Indeed, every Java program **must** have a *main class*.

2.5.1 The main class in Java

```
public class GenericProgramName {
  public static void main(String[] args) {
      // The magic starts here.
  }
}
```

The most important aspects to note (for now) in this example are:

- the curly braces {}
- the actual *main method* i.e.
  `public static void main(String[] args) {}`
- the actual *class* name for the overarching program i.e.
  GenericProgramName

In Java, units of code are enclosed in curly braces {}. The code between the first "{" and last "}" belongs together. Of course, this can, and often is, subdivided into further little bits. It may help to think of curly braces {} as boxes holding code. As with boxes, larger boxes can contain varied combinations of boxes within boxes - so too is true of curly braces cf.

{{{{}{}}}}

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

[1] Note: many introductions to Java would include the typical "Hello World" example as the first program. However, to build a practiced, conceptual understanding of Java as an *object oriented* language - and to reinforce and better practice *object oriented* thinking - this text is written to emphasize an *objects* first approach.