# Beginners' Guide to Object-Oriented Programming

Linxuan Ma

February 9, 2022

**Abstract**

This guide aims to layout the foundations and prerequisites for a student to proficiently program with object-oriented concepts. This guide intends to have a gradual learning curve and focuses on establishing a solid theoretical foundation before application to alleviate the "WHY THE HELL IS MY COMPILER SCREAMING AT ME" symptom that every programmer faces when starting off. Working knowledge with Java (minus the OOP part) is required.

## 1 Prerequisite: Objects

The core principle of object-oriented programming, or arguable any form of programming, is to abstract away the tedious implementation details and instead deal with simpler, higher-level concepts. One such example is the concept of function. Similar to its counterpart in mathematics, a function in the scope of computer science is a piece of *sealed* code whose implementation is irrelevant given that we are the caller and we know what it does conceptually. The function's *behavior* is the higher-level concept which we are interested in; the function's implementation detail is the lower-level that we abstract away. Function is an abstraction for the *procedure* of a program.

Similarly, *object* is an abstraction for the data of a program (loosely speaking). Instead of dealing with a plethora of arbitrary attributes and data in a program, we can instead group them into units called *objects*. This pattern allows us to have an array of `Person` instead of an abominable pile of `int[] age` and `String[] name`.

Having a bunch of "grouped" data is pretty useless by itself, as there wouldn't be a simple way to navigate and manipulate them. Therefore, instead of having arbitrary blobs of data with anonymous identity, Java associates each object with a *type*, which dictates the behavior and organization of the data within the object.

**Definition 1.1.** An *object* is a container for data, and has the following information:

- Class: a "blueprint" which specifies the behavior of the object
- Attributes: information contained in the object as fields

The concept of attribute is trivial; they are just what you put as fields in the class declaration of the object, e.g. `private int age`. The concept of type, on the other hand, requires much more attention, and is one of the core concept of object-oriented programming.

## 2 Theoretical Foundation of Types

"Type" might be a major source of confusion, only worsened by the more abstract ideas that "type" induces (e.g. polymorphism, variance). To simplify things, we will use simple mathematics concepts to illustrate the relation between types and objects, as well as the relation between types and types.

Types display remarkable resemblance of mathematical sets, and we can treat them as such in the scope of OOP. A type is just a set whose elements are objects. Some examples are:

$$\mathbf{T}_{\text{Integer}} = \{\dots, -1, 0, 1, \dots\}$$
$$\mathbf{T}_{\text{Boolean}} = \{\mathbf{true}, \mathbf{false}\}$$

We can check if an object is of a type by using the `instanceof` operator in Java. Such a check is equivalent to checking if an object is an element in the type's set in our mathematical notion:

$$x \text{ is of type } \texttt{Person} \iff x \in \mathbf{T}_{\text{Person}}$$

## 2.1 Notion of Equivalence

To formalize a set, we must define the notion of equivalence with regards to its elements. Unfortunately, there are two kinds of equivalence in Java (consider objects `a` and `b`):

1. `a == b`: compares whether the two references to objects are the same, i.e. if `a` and `b` refers to the *same* object
2. `a.equals(b)`: a programmer-defined equivalence behavior for the objects by overriding `Objectequals`