## A Brief Introduction to Object Oriented Programming

A number of languages are described as Object-Oriented, which means that the language is intended to allow the structure of a program to be neatly divided into discrete units that each define a specific portion of the program’s data and the logic that is closely associated with that data; these units are referred to as *classes*. To explain some terms courtesy of an overused analogy, consider a dog named Spot and a cat named Fluffy: Spot is an object whose class is dog, while Fluffy is an object whose class is cat. The term *class* represents the definition of the thing, while the term *object* represents one of those things; sometimes the term *instance* is used, as in: Spot is *an instance of* dog.

Now that the obligatory animal analogy has been made, let us instead consider a graphics system on a device or computer, starting with the concept of color. One representation of color is called RGB, in which three separate numeric values for red, green, and blue are mathematically combined into a single numeric value that represents the color; perhaps the simplest implementation of RGB values represents the three color channels as 8-bit integers, and the combined RGB value as a 24-bit integer. In an object oriented language, one can define a class named Color whose properties are Red, Green, Blue, and RGB. The notion of *encapsulation* allows the detail of how that information is stored inside a Color object to be hidden; for example, a Color could have three separate integers that it stores its red, green, and blue information in, or it could have one integer that it stores the combined RGB value in, or it could use some other representation of the data altogether. Regardless, from any point of view outside of the Color class, the *state* of a color object is easily represented by the *properties* of the object in a manner that is consistent with the description of what a color is, such that values of the Red, Green, Blue, and RBG properties are consistent.

Another valuable aspect of object oriented programming is *polymorphism*, which is when different classes of objects share some common programming *interface*, and thus are substitutable to some degree. Extending our previous example, a second class representing color could be created in a manner that would be substitutable in some situations for the first, by providing the same programming interface as the Color class provides, but potentially with a different data structure and implementation. For example, some JPEG and video formats use the YCbCr representation of color that is composed of a luma value (Y), and chroma values for both blue difference (CB) and red difference (CR). Since a YCbCr value can be converted to and from an RGB value, a second Color class (perhaps named YCbCrColor) could be defined that provides properties of Luma, BlueDiff, and RedDiff, but also provides properties of Red, Green, Blue, and RGB – just like the original Color class. Again, as with the previous example, encapsulation allows the class to hide the specific manner in which the object manages its internal information (also called its *state*) behind its programming interface, while exposing that information as both an YCbCr value *and* as an RGB value. Polymorphism allows an instance of the new YCbCrColor class to be used where the program was expecting an instance of the original Color class.

One way in which the YCbCrColor class could have been defined is through the use of *inheritance*, which is when a new class is defined by starting with the definition of an existing class, and then adding things to it. Inheritance is quite powerful, particularly because it allows the re-use of existing class definitions, and because it provides a natural means of supporting polymorphism. In our example, if YCbCrColor is inherited from Color, then it would automatically inherit all of the programming interface of Color, and an instance of YCbCrColor could be transparently used in place of an instance of Color, because YCbCrColor *is a* Color. However, as powerful as inheritance may seem, it has often been perceived to be overused and even abused as a tool; as a result, developers have started to “favor object composition over class inheritance”[[1]](#footnote-1).

While inheritance provides an “*is a*” relationship, composition provides a “*has a*” relationship. For example, instead of the YCbCrColor class inheriting from the Color class, it could simply have a property of the Color class; in other words, the programming interface for YCbCrColor could include a way to obtain a separate object that *is a* Color value, just like it includes a way to obtain Luma, BlueDiff, and RedDiff values.

A third option is to use interface inheritance, which allows the programming interface from an existing class to be re-used while not inheriting other aspects of the existing class definition, such as the manner in which data is structured and managed. For example, a programming interface called Color could be defined that has a single property RGB, whose value is in the form used natively by the device, such as a 24- or 32-bit integer or floating point value. Any number of class definitions could then *implement* the Color interface, thus allowing instances of those various classes to be utilized any time a Color is required. For example, the Color class from our original example could be renamed as RGBColor, and both it and the YCbCrColor class could be made to implement the new Color interface. Similarly, the interface could be implemented by additional class definitions, such as a YUVColor class (the color value used in PAL televisions), and a YIQColor class (the color value used in NTSC televisions).

Conceptually speaking, a programming interface is composed of the accessible *properties* (state) of a class and the *methods* (behavior) that the class exposes; collectively, these are the *members* of the interface. Some programming languages allow any class to act as a substitute for another class, as long as it has the same members as the class that it is substituting for; this is an example of *dynamic typing*, and is often specifically referred to as *duck typing* because “if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.” Other languages require that a class explicitly declare the names of the programming interfaces for which it can act as a substitute; generally speaking, this is an example of *static typing*. However, it is important not to approach terms such as static and dynamic typing too rigidly, because these concepts represent a continuum of possibilities, as opposed to some single binary choice. Furthermore, every programming language has many different aspects, each of which can be located at a completely different point on that continuum.

In object oriented programming, each class is a specific data type; in our examples, each of the various color-related classes and programming interfaces is its own separate data type. When a programmer wishes to create a class that operates on some yet-to-be-specified second class, that second class can be represented by a place-holder, which is called a *type parameter*. A class that has one or more type parameters is called a *parameterized type*, or simply a *generic type*. We have already discussed “*is a*” and “*has a*” relationships; a generic type supports an “*of a*” relationship. Consider the array, which is the most basic of data structures, supported even by assembly languages. An array is a contiguous sequence of a specific number of elements of a specific type, each of which can be loaded from or stored into the array using the element’s index, which is an absolute position in the array. While static typing makes it possible to specify that one must pass a Color to particular method, consider a method that allows any number of Color objects to be passed to it; without generic types, the programmer would have to define the method to accept *any* Array (whose contents are of an unspecified type), or alternatively could create a specific type – perhaps ArrayOfColors – that can hold a number of different colors. A generic type is an alternative to creating a separate specific type (such as ArrayOfColors) for each separate “*of a*” relationship; instead, a single type is defined with a place-holder, such as Array<T> which uses T as a place-holder for some unspecified type. Using a generic type would allow the method in our example to be defined to accept a parameter of type Array<Color>, which is read as “an Array of Color”. The benefit of generic types is the extension of type safety (and any other facilities that rely on or can benefit from type knowledge) to any classes that support one or more “*of a*” relationships. This is also referred to as *parametric polymorphism*, which refers to both the *parameterized type* aspect, and polymorphism, which is a direct benefit of using parameterized types (generic types) as the basis for generic programming.

One of the valuable capabilities provided by inheritance, and perhaps one of the causes for the overuse of inheritance, is the ability to define state and behavior in one class that is then inherited by many different classes. Elimination of redundant code (*a la* “cut and paste”) is a noble goal indeed, and is at the core of the “Don’t Repeat Yourself” (DRY) principle, which states: “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system[[2]](#footnote-2).” In addition to inheritance, you can see this principle at work in programming interface definitions, polymorphism, and generic types. The concept is well described by Steve Smith[[3]](#footnote-3):

Of all the principles of programming, Don't Repeat Yourself (DRY) is perhaps one of the most fundamental. The developer who learns to recognize duplication, and understands how to eliminate it through appropriate practice and proper abstraction, can produce much cleaner code than one who continuously infects the application with unnecessary repetition.

Every line of code that goes into an application must be maintained, and is a potential source of future bugs. Duplication needlessly bloats the code-base, resulting in more opportunities for bugs and adding accidental complexity to the system. The bloat that duplication adds to the system also makes it more difficult for developers working with the system to fully understand the entire system, or to be certain that changes made in one location do not also need to be made in other places that duplicate the logic they are working on.

Repetition in logic can take many forms. Copy-and-paste[..] is among the easiest to detect and correct. Many design patterns have the explicit goal of reducing or eliminating duplication in logic within an application. In fact, the formulation of design patterns themselves is an attempt to reduce the duplication of effort required to solve common problems and discuss such solutions.

A powerful mechanism for achieving DRY in object oriented programming is the *mix-in*, which in many ways is like a class definition, but one that can be “mixed into” (added to) other class definitions. Using the concepts of generic programming, a mix-in *applies to* some yet-to-be-specified second class that provides a specified programming interface, allowing the logic of the mix-in to be constructed around the abilities of that programming interface. Then, any time that logic is required in another class, the mix-in can be applied to that class; while various other mechanisms provide “*is a*”, “*has a*”, and “*of a*” relationships, a mix-in satisfies the “*needs a*” relationship. Continuing our color example, some graphics systems support the notion of color transparency, which is sometimes expressed in the range of 0% to 100%, but is often encoded as an 8-bit value prepended to a 24-bit RGB value, creating a 32-bit integer value. A ColorTransparency mix-in can be defined that applies to any Color, adding a Transparency property to hold a value representing the transparency level, and overriding the RGB property to incorporate the transparency information into the resulting value.

A mix-in that does not define its own state is called a *trait*; a trait is simply a stateless mix-in. As such, a trait can only be used to inject logic (behavior definition) into a class. Using the color example, an Inverting trait can be defined that applies to any Color; the Inverting trait adds an invertedRGB() method that calculates and returns an RGB value that is the inverse of the RGB property value.

This “brief” introduction was not intended to be an exhaustive explanation of object oriented programming; hopefully, it has introduced a number of terms and concepts, and with enough context to provide a basis for understanding the following constructs of the Ecstasy language.

1. From the book “Design Patterns: Elements of Reusable Object-Oriented Software”, GoF *et al* [↑](#footnote-ref-1)
2. From the book “The Pragmatic Programmer”, by Andy Hunt and Dave Thomas [↑](#footnote-ref-2)
3. With edits, licensed using [Creative Commons Attribution 3](mailto:https://creativecommons.org/licenses/by/3.0/us/) by Steve Smith [↑](#footnote-ref-3)