SnaX

Tutorial

Tutorial 2: Object Orientated Programming!

Revision 3

Introduction

In this tutorial, we are going to have a look at the *object-oriented programming model* of SnaX. We are going to create a very simple program using some basic object-oriented concepts like *classes*, *inheritance*, *data members*, *member functions*, *function overriding*, and *class instances* (*objects*). Basically, we are going to learn how to do the following things in SnaX:

- Create new classes.
- Create static and nonstatic member functions.
- Call functions.
- Create instance data.
- Create instances and modify their local data members.
- Use the class diagram to implement inheritance.
- Override functions and call overridden base class functions.

This tutorial is not really about teaching you object-oriented programming in general. If you are new to this, google will probably help you get started. If you are new to SnaX, reading Tutorial 1 first is recommended. Don't forget to have a look in the user manual as well! The essentials to complete this tutorial are highlighted in yellow!

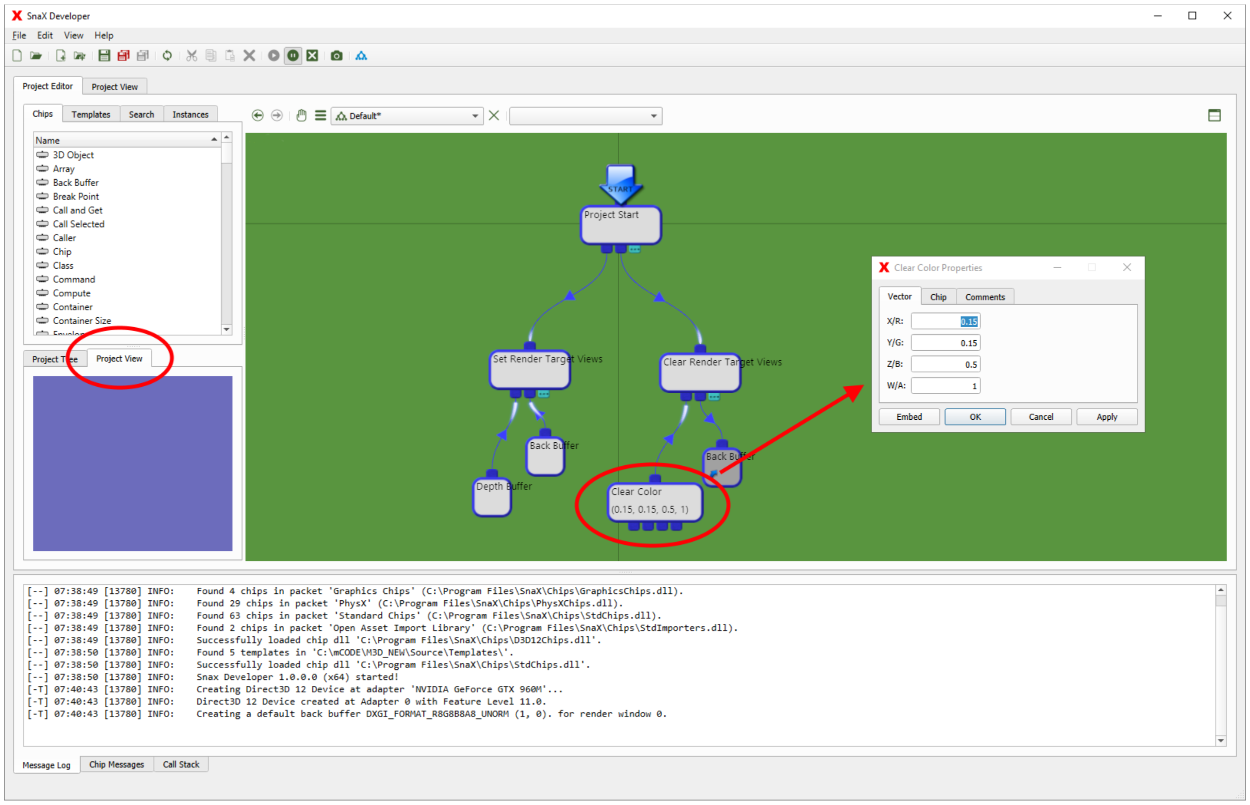
Tutorial

Step 1

The first thing we have to do is to start up SnaX Developer. By default, it will start with a simple program that renders the Utah teapot. Open one of the project views to verify this.

We are not going to do any fancy rendering in this tutorial, but to teach you the basic of object-oriented development with SnaX, we are going to have a look at the color that clears the screen. Start by removing most of the logic from the default program. Only the code for setting and clearing the render target should remain, as seen in Figure 1.

Open the small project view in the down, left corner of the Developer again. The teapot is gone, and the view is cleared to a bluish color as seen in the figure. We are now going to change the background color of the view. Have a look in the editor view and find the chip named 'Clear Color'. This is a `Vector` holding four floating-point components, in this case representing the red, green, blue and alpha color components in the range 0 to 1, used to clear the back buffer. Open its property dialog, try changing the different values, and see how this affects the color of the project view.



X/R: 0.15
Y/G: 0.15
Z/B: 0.5
W/A: 1

```
[--] 07:38:49 [13780] INFO:    Found 4 chips in packet 'Graphics Chips' (C:\Program Files\SnaX\Chips\GraphicsChips.dll).
[--] 07:38:49 [13780] INFO:    Found 29 chips in packet 'PhysX' (C:\Program Files\SnaX\Chips\PhysXChips.dll).
[--] 07:38:49 [13780] INFO:    Found 63 chips in packet 'Standard Chips' (C:\Program Files\SnaX\Chips\StdChips.dll).
[--] 07:38:49 [13780] INFO:    Found 2 chips in packet 'Open Asset Import Library' (C:\Program Files\SnaX\Chips\StdImporters.dll).
[--] 07:38:49 [13780] INFO:    Successfully loaded chip dll 'C:\Program Files\SnaX\Chips\D3D12Chips.dll'.
[--] 07:38:50 [13780] INFO:    Found 5 templates in 'C:\mCODE\M3D_NEW\Source\Templates\'.
[--] 07:38:50 [13780] INFO:    Successfully loaded chip dll 'C:\Program Files\SnaX\Chips\StdChips.dll'.
[--] 07:38:50 [13780] INFO:    Snax Developer 1.0.0.0 (x64) started!
[-T] 07:40:43 [13780] INFO:    Creating Direct3D 12 Device at adapter 'NVIDIA GeForce GTX 960M'...
[-T] 07:40:43 [13780] INFO:    Direct3D 12 Device created at Adapter 0 with Feature Level 11.0.
[-T] 07:40:43 [13780] INFO:    Creating a default back buffer DXGI_FORMAT_R8G8B8A8_UNORM (1, 0). for render window 0.
```

Figure 1 Setting the color to clear the viewport.

Step 2

We are now going to take this one step further and get the clear color from a *static function* in another *class*. The first thing we have to do is to create a new class! There are several ways to do this, for example by right clicking the document called ‘Not Saved’ containing the Default class in the project tree panel, and select ‘New Class…’ from the menu, as seen in Figure 2. The nice thing about this method is that the new class is added to the same document as the Default class. When saving the project, we only get one project file instead of two.

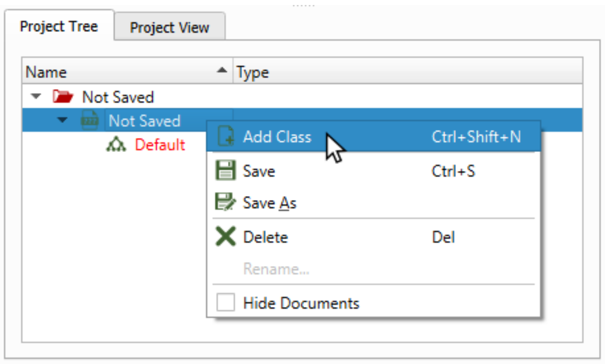


Figure 2 Adding a new class to an existing document.

Okay, name the new class for ‘Color’ and click OK in the dialog box. The new, empty class should now be visible in the editor view. Let’s go ahead and add a few chips to it! Look in the list of chips and add one `Vector` and four `Values`. Connect the `Values` as children to the `Vector`, and name them ‘Red’, ‘Green’, ‘Blue’ and ‘Alpha’, respectively, using the chips’ property dialogs. Open the `Vector`’s property dialog, name the chip GetColor and make it a static function, as illustrated in Figure 3. Making a chip a *function*, gives it the special *property* of being remotely accessible by a `Function Call`. As any programmer should know, a *static member function* is called on the class itself, and does not need a *class instance*.

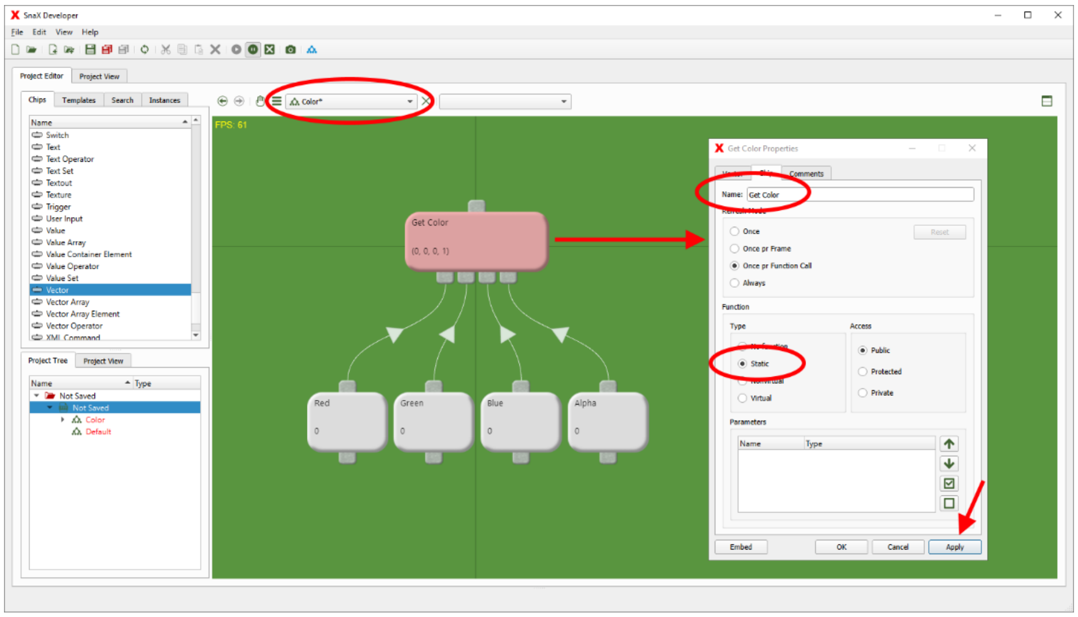


Figure 3 Creating a static member function.

Go back to the Default class, find the 'Clear Color' chip, and remove it. Have a look in the project tree, expand the Color class, and drag the GetColor function into the Default class, as seen in Figure 4. This creates a `Function Call` targeting the static GetColor function of the Color class. Now, link it to where the 'Clear Color' chip used to be connected, and open the project view. Switch back to the Color class again and try modifying the `Values` connected to the GetColor function. Does changing the values change the color of the project view? Cool, you've just learned how to call a static member function! Note how a *function* is just a property that can be given to (almost) any chip, while a *function call* is implemented as its own type of chip, namely the `Function Call`.

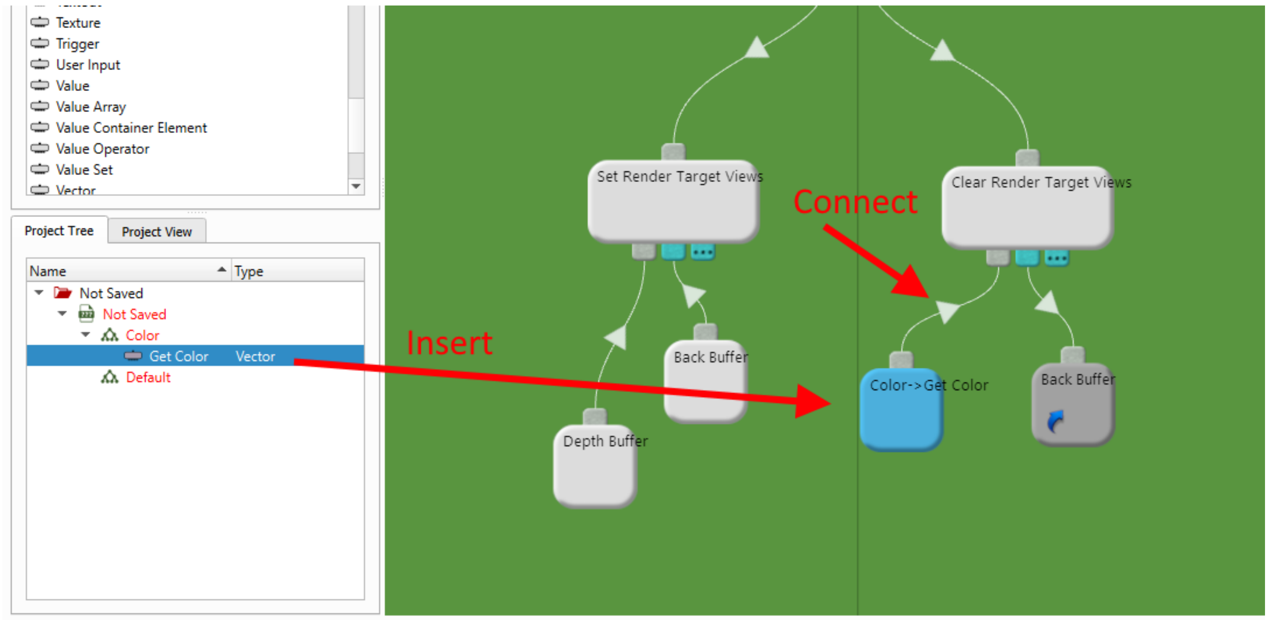
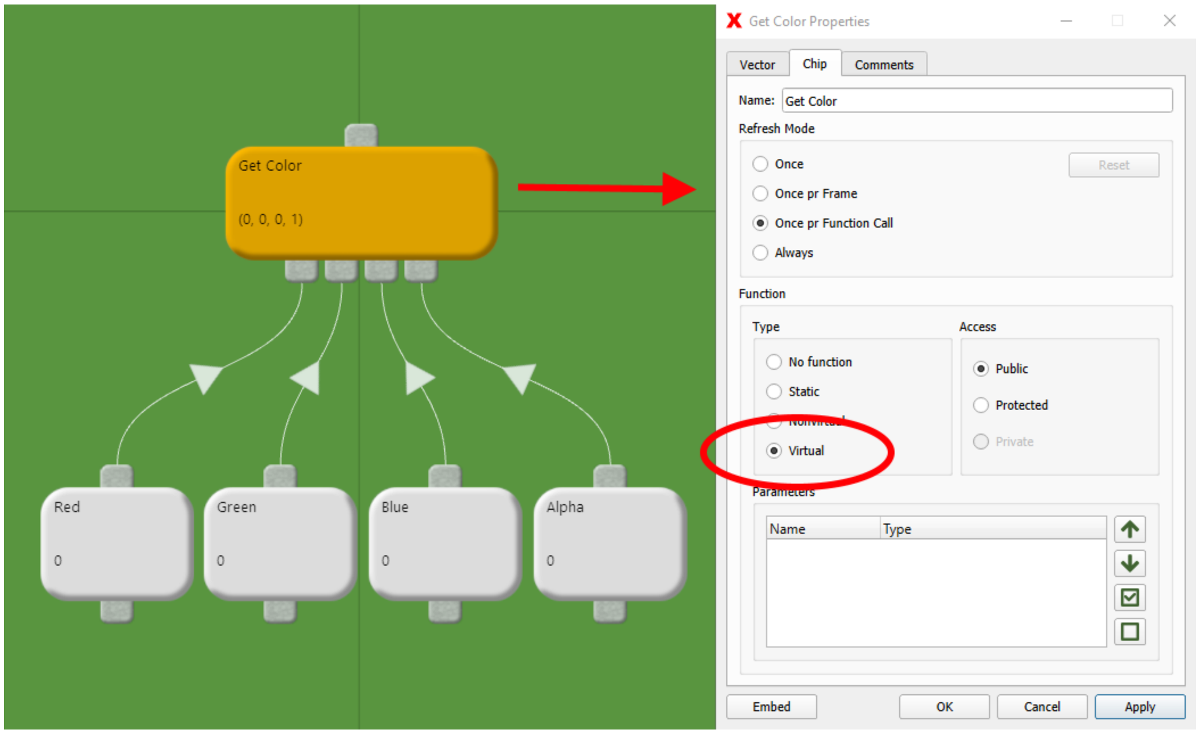


Figure 4 Adding a Function Call.

Step 3

As mentioned, a *static* function does not require a class instance. Looking at the function call, we notice that it does not have any child connectors. We are now going to change the function from being static to *virtual*. A virtual function, like a *non-virtual* function, requires a class instance when being called. Unlike a non-virtual function though, a virtual function can be *overridden* in *deriving* classes. More on that later! Changing the function type is very simple; just go to the function's property dialog and select 'Virtual' instead of 'Static' in the function configuration, as indicated in Figure 5. The color of the chip will also change to indicate the current function type!



Name: Get Color

Refresh Mode: Once / Once pr Frame / [✓ Once pr Function Call] / Always

Function

Type: No function / Static / Nonvirtual / [✓ Virtual]

Access: [✓ Public] / Protected / Private

Figure 5 Changing to a virtual function.

Have a look at the function call again. A child connector has appeared! This is where we are going to connect the *class instance* for which we are going to call the function upon! Go to the list of chips, find an `Instance Ref` chip, drag it into the Default class, and connect it to the function call as illustrated in Figure 6.

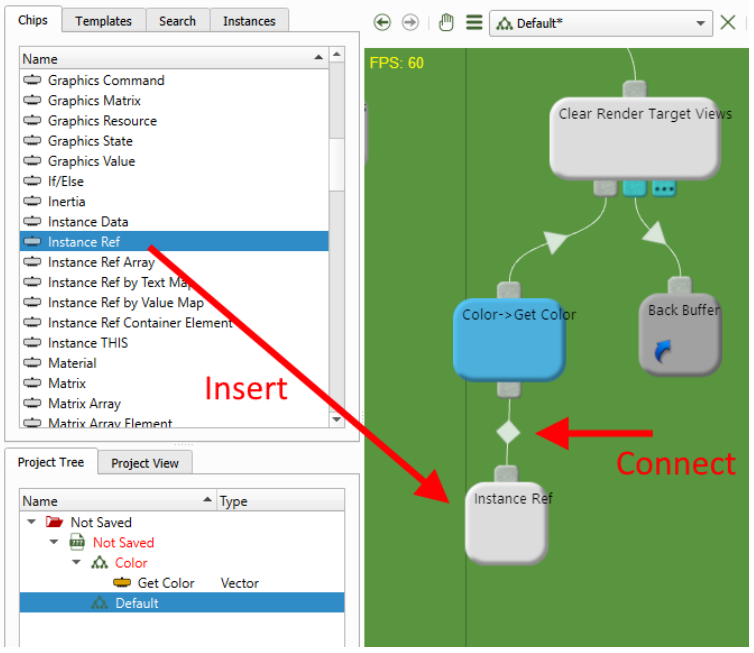


Figure 6 Adding and connection an Instance Ref chip to the function.

We just created a class instance *reference*, but we did not assign any *instance* to it! This is just like a *null pointer* in C++, except that the program does not crash if we try calling a function upon it! Just try! Open the reference's property dialog, and create a new instance of the Color type, as seen in Figure 7. Open the project view again and confirm it still work as it did using a static function. You can still change the color components in the Color class. Try adding a second `Instance Ref`, assign it a new Color instance and link it to the function call, discarding the old link. It will give the same color as the other instance, right? Wouldn't it be nice if we could make the two instances return different colors? Yes, that's exactly what object-orientation is about, and to achieve this we need to add some *instance data* to our Color class!

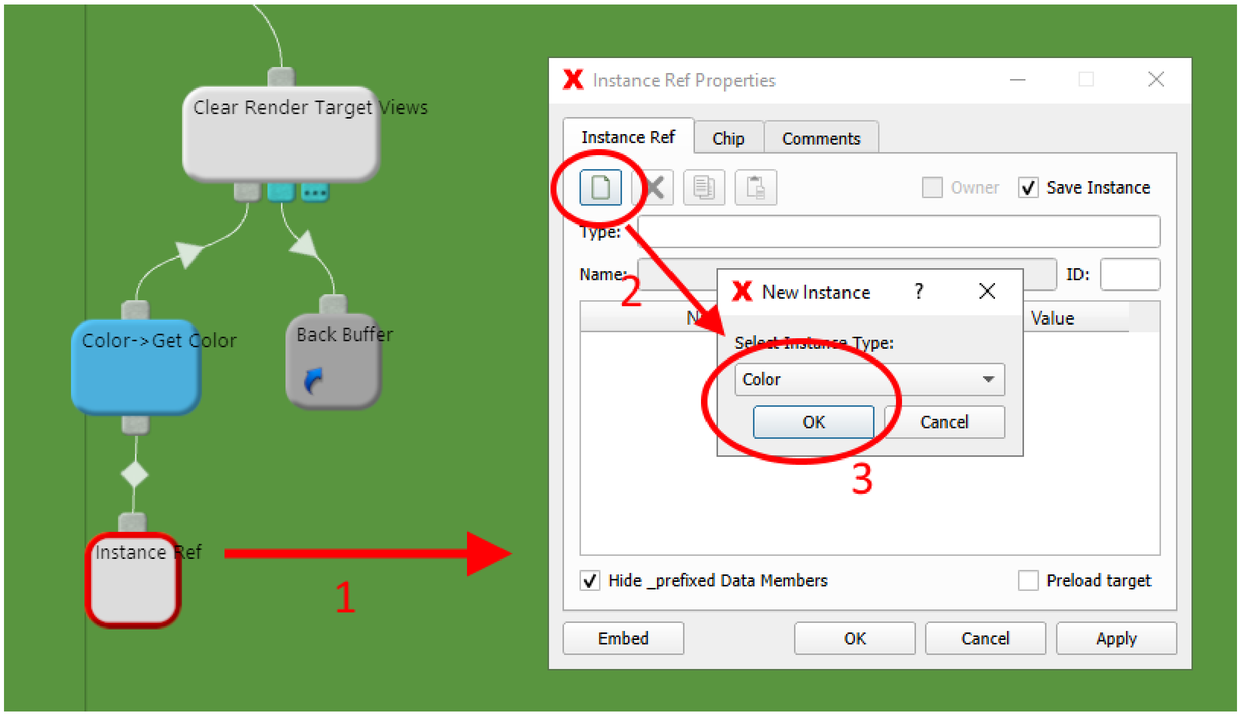


Figure 7 Creating a new 'Color' instance.

Step 4

All chips you can see in a class are in fact *static data members* of the class. This means that they are common to all instances of the class, just like we realized in the previous step. To create data, or chips, that are unique to each instance, we need to use the `Instance Data` chip. We want to make the `Values` (Red, Green, Blue and Alpha) in the Color class, data members of each instance we create. To do so, we can convert each of them to `Instance Data` by right clicking each chip, then select 'Convert to Instance Data', as illustrated in Figure 8. Note that the *state* the chip currently has when converting, will be the default state for each new instance of the chip. In our case, the state is the floating-point value each of the values hold. If we want each new instance of Color to return a green color by default, we should make sure the Red, Green and Blue chips hold the values 0.0, 1.0 and 0.0 respectively when converting them. Of course, we could change this later using each `Instance Data`'s property dialog!

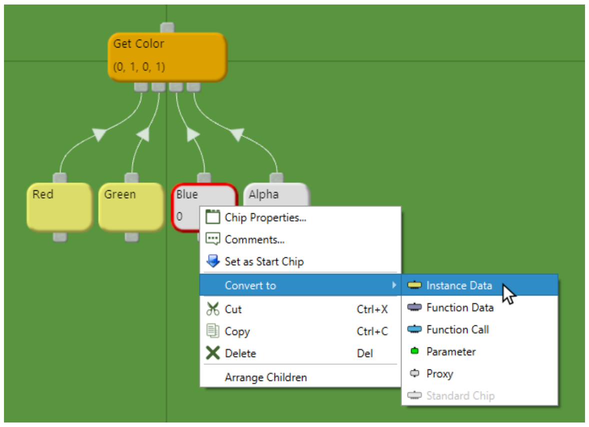
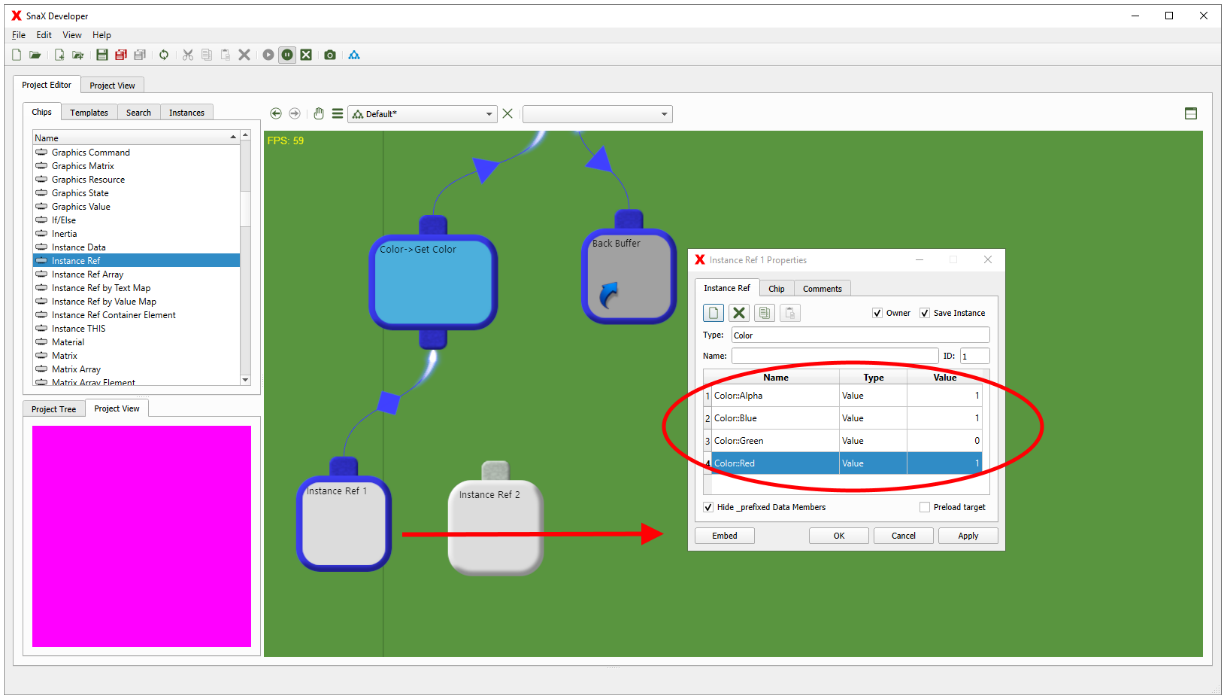


Figure 8 Converting chips to Instance Data chips.

Note how the four values in our class got a yellow color when you converted them. Like the `Function Call`, an `Instance Data` is this special type of chip we give its own color to help us identify them. Now, go back to the Default class and open the property dialog for the two `Instance Ref` chips you created earlier. Note how each instance now has four members named 'Red', 'Green', 'Blue' and 'Alpha' in the data member table as highlighted in Figure 9. Their values should be equal to the values of the old chips at the time we converted them to instance data. Try giving each instance a unique color by changing the values in the table, and then open the project view. Its color should reflect the setting in the instance currently linked to the function call. Now, link the other instance instead! Did the color of the project view change? Yes? Great! Each instance's data is accessed through the respective `Instance Data` chips in the Color class. These chips will return the data from the instance the function was called upon. The `Instance Data` chips are just *physical substitutes* for the actual instance data that are stored inside a class instance.



Type: Color
Name:
ID: 1

	Name	Type	Value
1	Color::Alpha	Value	1
2	Color::Blue	Value	1
3	Color::Green	Value	0
4	Color::Red	Value	1

[✓] Hide _prefixed Data Members
[ ] Preload target

Figure 9 The data member table of a Color instance.

Step 5

We are now going to have a look at *inheritance*! This is a key concept in object-oriented programming, and just like C++, SnaX supports *multiple* inheritance! Inheritance is when a class is *based on*, or *inherits*, another class to reuse and extend its functionality.

Let's begin by creating a new class. You have already learned how to do this! Call the new class for ScaledColor. Add a `Value` chip named 'ScaleFactor', set it to 1.0 and convert it to an `Instance Data`. We are now going to let the new class inherit the Color class. To do this we have to open the *Class Diagram* from the toolbar or the main menu. This blue workspace contains all classes in our project. Each chip represents a class, and their child connections represent *base classes*; the classes they inherit, or *extend* as we say. We are going to let ScaledColor inherit the Color class. Do this by dragging a new link from the ScaledColor's child connector to the top connector of Color, as illustrated in Figure 10. You are free to arrange the chips as you like to get a clean and organized class diagram. Those familiar with *UML class diagrams* will notice that the class diagram of SnaX is upside down!

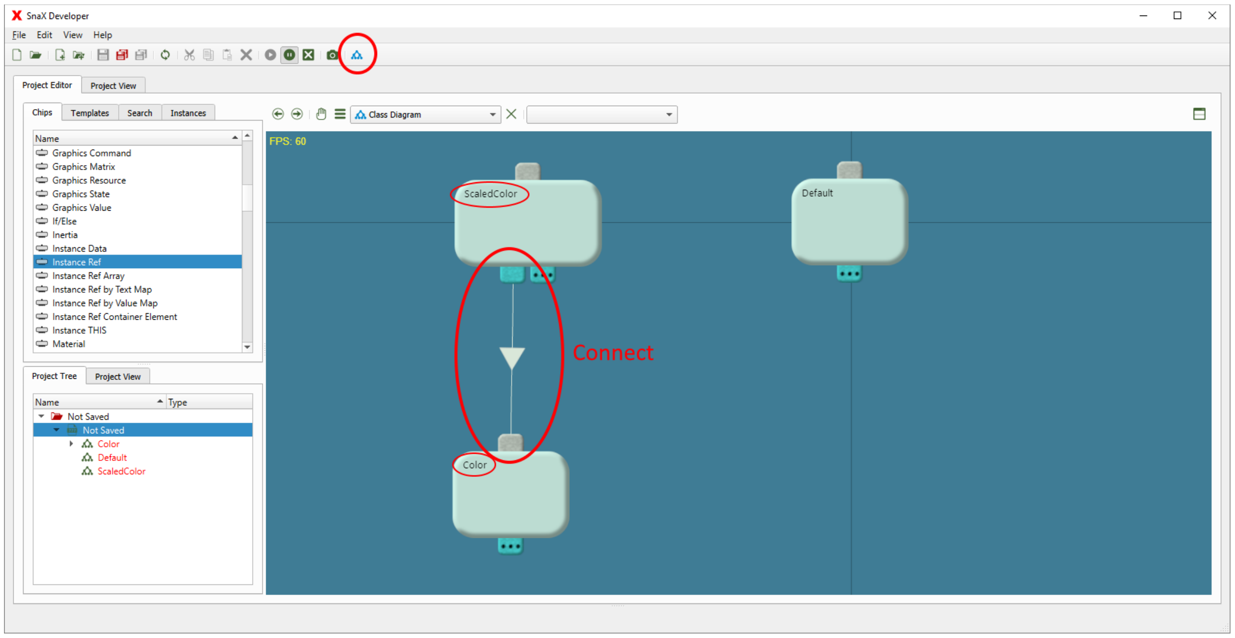
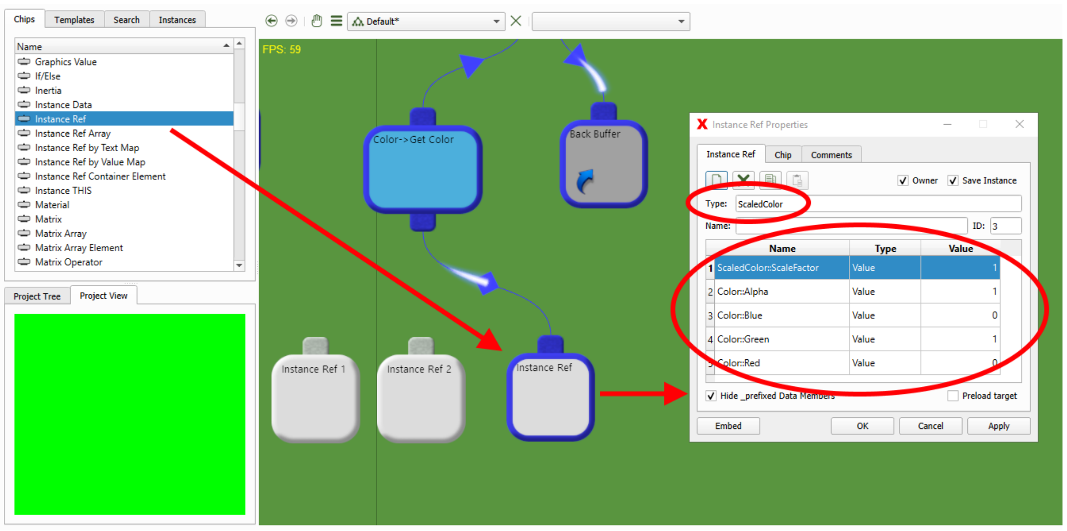


Figure 10 The class diagram of SnaX. ScaledColor inherits the Color class!

By inheritance, the ScaledColor class is now also a *type of* a Color class. Any function that accept a Color instance will also accept a ScaledColor instance, a concept known as *type polymorphism*. Go ahead and add a new `Instance Ref` to the Default class, and create a new instance of ScaledColor. You will notice that the data table contains the four data members of the Color class in addition to the ScaleFactor member we added to the ScaledColor class, as seen in Figure 11. Give it some random color and link the `Instance Ref` chip to the function call. Open the project view and confirm it work exactly as for the Color instances! If we did not let ScaledColor inherit Color, an error message would be generated instead. So far, we have not really added any new functionality to the new class, and the ScaledColor member is not used at all. That's what we are going to do something about now!



✓ Owner ✓ Save Instance

Type: ScaledColor

Name: ID: 3

	Name	Type	Value
1	ScaledColor::ScaleFactor	Value	1
2	Color::Alpha	Value	1
3	Color::Blue	Value	0
4	Color::Green	Value	1
5	Color::Red	Value	0

✓ Hide _prefixed Data Members ☐ Preload target

Figure 11 A ScaledColor instance also contains data members from the Color class.

Step 6

The last concept we are going to look at is *function overriding*. As we saw in the previous step, calling GetColor of Color on a ScaledColor instance worked just like earlier. Remember that we made this a virtual function, one that can be *overridden* in a derived class. This means that if we create an identical function in our ScaledColor class, this function will be called instead of Color's version of the function when we pass an instance of ScaledColor to our function call. This happens is even if the function call is still targeting Color's version of the function! That is called *function overriding*, and we are going to give it a try now!

We are now going to override Color's GetColor function in our ScaledColor class. The easiest way to do this is to just copy the function (select it, then press Ctrl+C), and then paste it (Ctrl+V) in the ScaledColor class, as illustrated in Figure 12. That's it! Linking a ScaledColor instance to the function call in our Default class will no longer call GetColor in the Color class because it is overridden; the function in ScaledColor is called instead! If you connect one of the Color instances again, the original function is of course again called. This is an important point, because it means that inheriting a class does not change or break the existing behavior of the base class or any of its instances!

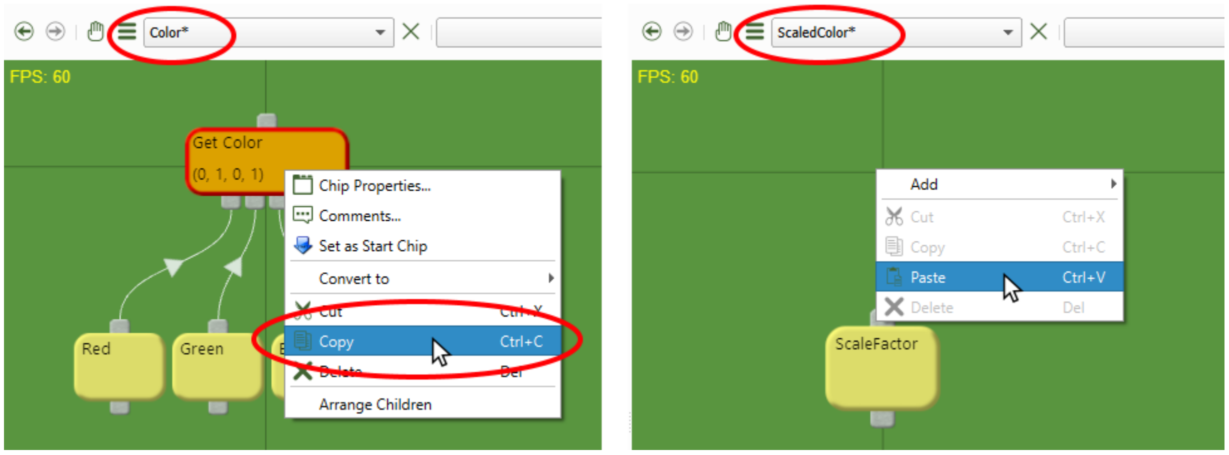


Figure 12 Overriding a function by copy and paste.

Still, we have not added any functionality to the function in ScaledColor. We want this new function to return the color from Color's GetColor function multiplied by our ScaleFactor. The first thing we have to do is to call GetColor of the Color class to get the original color. The easiest way to do this is to find Color's GetColor function in the project tree and drag it into the ScaledColor class while holding down the *Shift* key. This will create a function call that is set to be '*called by name*'. Like for a static function call, it does not have a child connector for an instance, but it will implicitly use the instance we are currently working on, calling the exact function it is targeting, ignoring overriding. Now, insert a `VectorOperator` to the class, open its property dialog, and select 'Multiply Vectors' from the 'Vector Operations' list. Link the newly added function call to the first child connector, and a new `Vector` to the other. Take the ScaleFactor chip and link it to the first three ('Red', 'Green' and 'Blue') child-connectors of the `Vector`. This implements the scaling operation we want.

The last step is linking the scaling-logic we created to the function. Since the function is a standard `Vector`, this is not directly possible since a `Vector` accepts four `Value`s, and not a `Vector` in its child connections. The solution is to convert the function into a `Proxy` chip as illustrated in Figure 13 – Then connect the `VectorOperator` to the child connector. A `Proxy` is just a dummy, a powerful such, forwarding whatever is connected to its child connector, in our case the `VectorOperator`, to anyone who calls the function. As a side note, the `Proxy` is the base for many other powerful chips like the `Instance Data`, `Function Call`, `Switch` and a few more. They all have one thing in common: They can *take* any type, and will *forward* another chip to whoever calls it. Like the `Instance Data`, a `Proxy` acts as a physical substitute for another chip, which for the `Proxy` itself is just the chip connected to its sole child connector. In our case, it takes the type of a `Vector`, but it is not a `Vector` itself, and we can use it just like any other type of `Vector`. This feature is one of the corner stones in the programming model of SnaX, and is an extremely powerful concept!

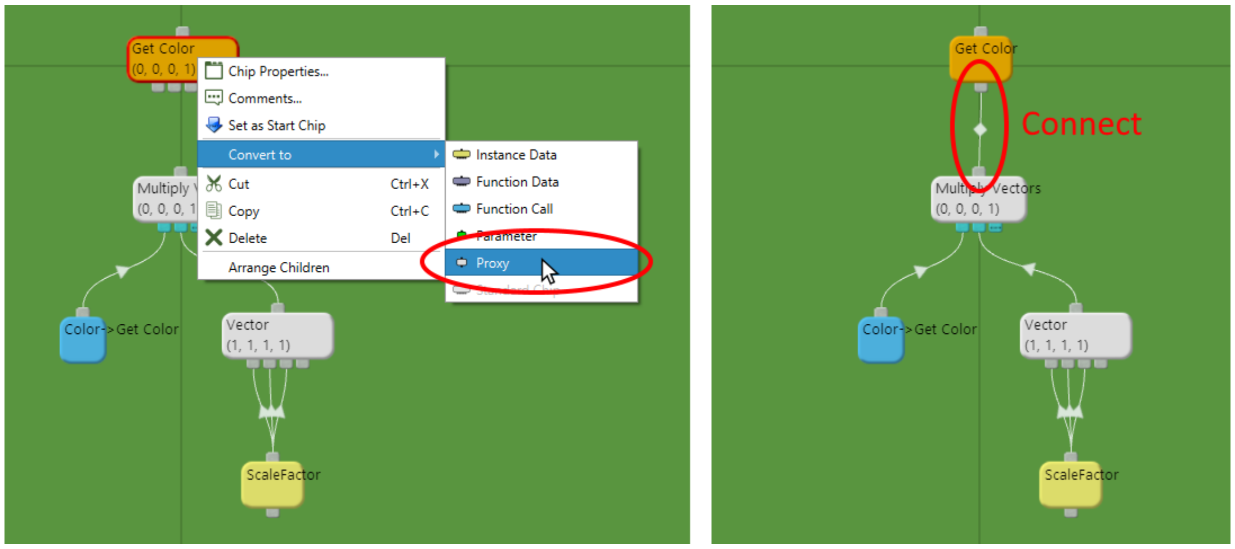
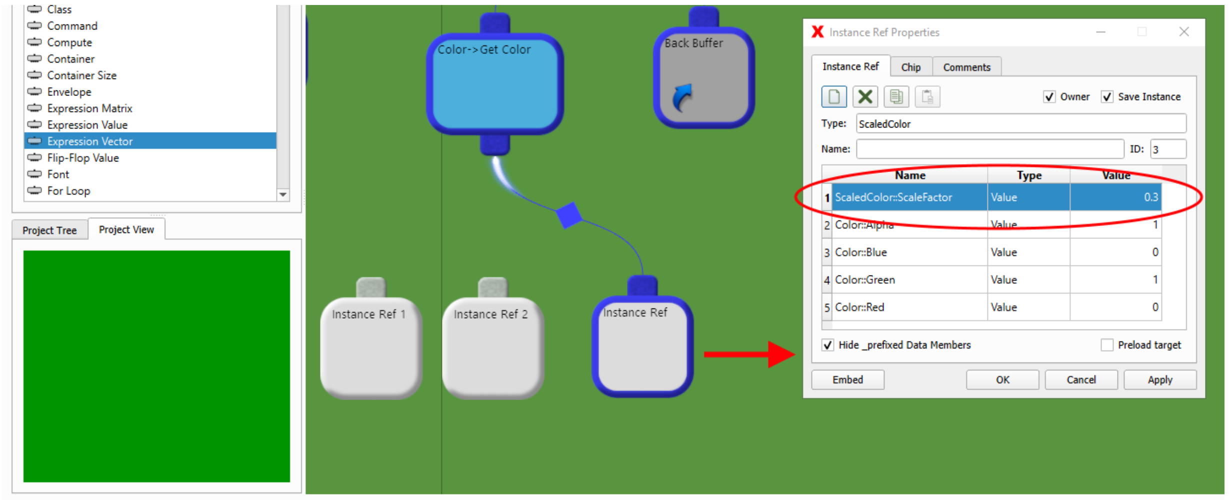


Figure 13 Converting the function to a Proxy chip.

Finally, open the project view again with the ScaledColor instance connected to our function call in the Default class. Like Figure 14 illustrates, try changing the instance's ScaleFactor member and see what happens. The intensity of the color in the project view should change! This is, as you hopefully should know by now, because the GetColor function is being called in ScaledColor because the function is overridden. This function calls the base implementation in the Color class, and multiplies the color it returns by the scale factor.



☑ Owner ☑ Save Instance

Type: ScaledColor

Name: ID: 3

	Name	Type	Value
1	ScaledColor::ScaleFactor	Value	0.3
2	Color::Alpha	Value	1
3	Color::Blue	Value	0
4	Color::Green	Value	1
5	Color::Red	Value	0

☑ Hide _prefixed Data Members ☐ Preload target

Figure 14 ScaledColor uses the ScaleFactor to change the intensity of the color.

That's it! You have now learned the very basic of object-oriented programming in SnaX. There are still a lot more to learn about this, like how to use *function parameters*, *data structures* (like `Instance Ref Arrays`), *multiple inheritance*, *access modificators* and so on. Have a go and see what you can figure out by yourself, using this tutorial as a starting point!

Enjoy!