

The Language

This chapter will explain the basic building blocks of a program patched in VL. We'll focus on what is expressible and what every programming language is about: *telling the computer what to do*.

We'll only briefly touch how to edit or even refactor programs. See "The Development Environment" for getting started with the UI. We'll only discuss some basic nodes and data types that are central or work for illustration purposes. For more about nodes see "Libraries".

Looking at Things

A rough overview laying out the groundwork for later chapters that look **into** things.

Looking into Things

- *More On Data*: Different shapes of data and how to work with data.
- *Patches as Defining Constructs*: How to create nodes that others can use.
- *Inside Patches*: All about data flow and what to do on the canvas.

Looking at Things



Data and Data Hubs

"hello world"

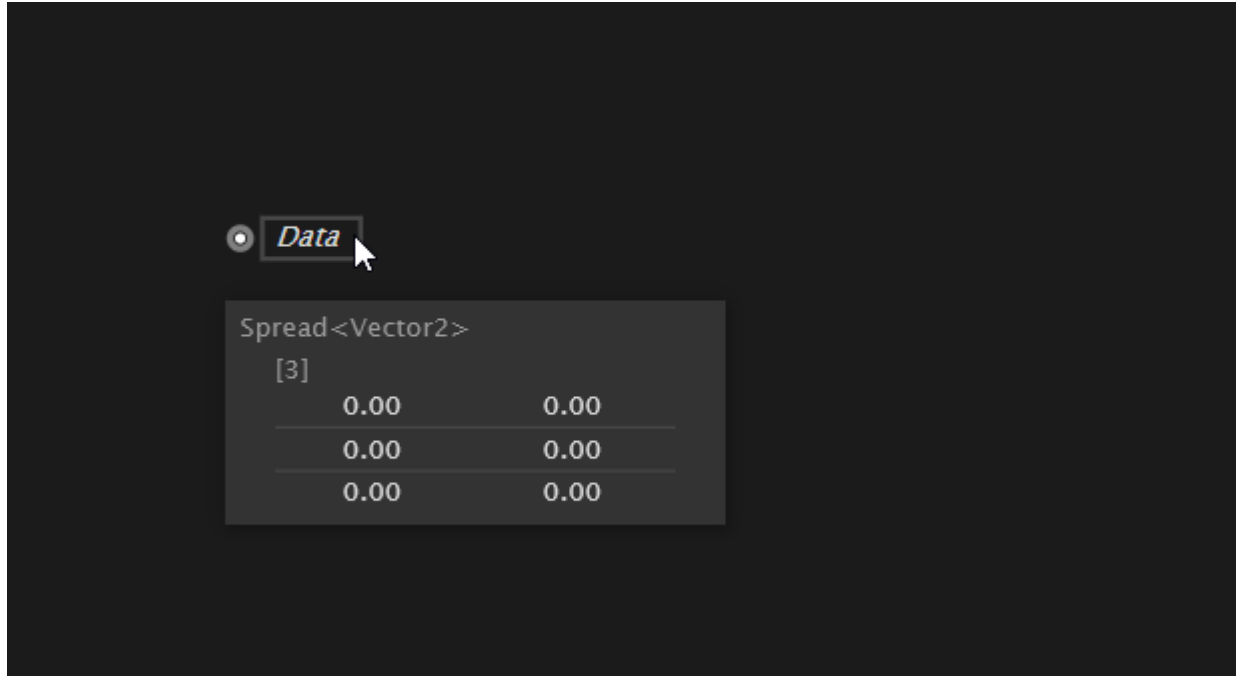


Data is everywhere.

And programming pretty much is all about dealing with data.

Programming is creating a description for a machine how to deal with some potentially given data in the future. In a rapid prototyping system like this you often also have actual current data at hand and are able to inspect that data here and there.

- [Hover on Data](#)
- [Toggle Dataview/Editor](#)



inspecting data by hovering over data-hub

Tip

The white dot in the data-hub indicates that the datatype for this one is explicitly set.



(Spread<Vector2> in this case)

However the program you are building is typically concerned with **any data** that may arrive when the program will run **in the future**.

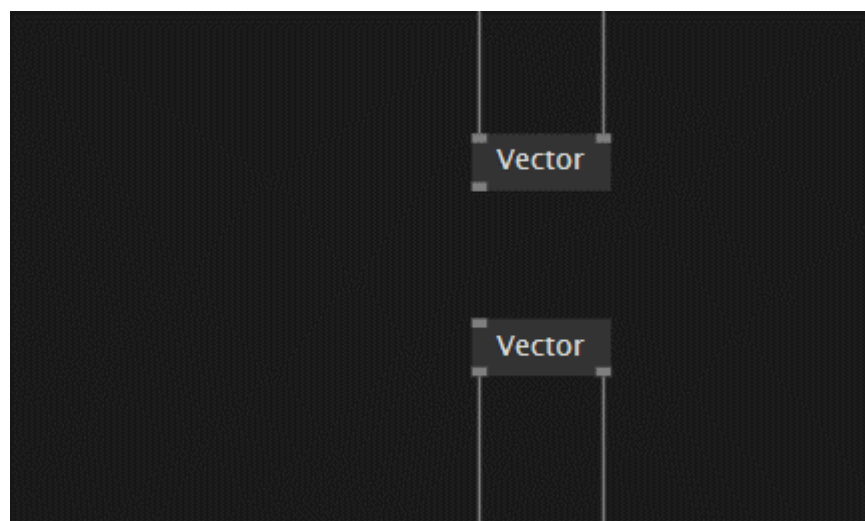
So you'll be thinking all about

- accepting *data* & computing with *data*
- extracting features of *data*
- filtering *data* & yielding *new data*
- feeding back *data* & saving *data*
- visualizing *data* & sending *data*
- inspecting *currently available data*
- describing what to do with *data available at any given point in time*
- abstracting over *specific data* and thinking more in *possible types of data*

For the latter have a look at [Data and DataTypes](#). For now let's just go ahead and make up a few terms: data hubs, data sources, data sinks.

A data hub is something that you can connect to. It's either a data source  or a data sink . A data source is able to give you data that you can work with. A data sink would love to be given some data from you. Both let you link to.

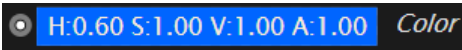
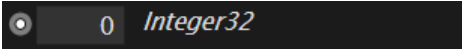
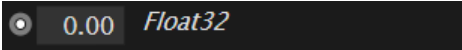
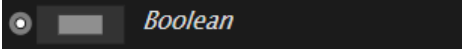
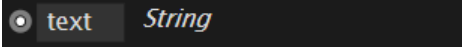
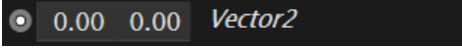
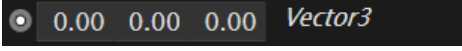
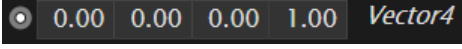
Well, actually data flow programming is mainly about linking data sources to data sinks. The idea being that the data flows from a data source into a data sink.



data source getting linked to a matching data sink

Data and Data Types

Let's look at some more data:

Type	Data-Pad
Color (RGBA)	 <i>Color</i>
Number (Integer32)	 <i>Integer32</i>
Number (Float32)	 <i>Float32</i>
Boolean (true or false)	 <i>Boolean</i>
String (aka a text)	 <i>String</i>
Vector2	 <i>Vector2</i>
Vector3	 <i>Vector3</i>
Vector4	 <i>Vector4</i>

You get the picture. All of this above is called data. But still all this data differs a lot and often you need to know where to expect which "type of data". Luckily people refer to the "type of data" as the "data type", which just captures this very basic idea of ordering the mess by separating all possible data into different types of data, which themselves refer to all possible data of that type of data.

E.g. at a particular data source in our patch we expect to get a hand onto some data of type color - in short a color. As soon as we know that data type we know that of all possible data in the world there is just small subset of possible data that may arrive at this particular data hub. The color may be red or blue or even purple, but it has to be a color by all means. If we always needed to think about any possible type of data we'd need to have a lot of fallback solutions in our code for the case it actually is random data that we didn't expect...

Since data and data types are everywhere people started calling a data type just the "type" skipping the redundant "data" part.

Instances

Data of a certain data type also is called an instance of that data type.

For the vvvv user:

In VL we even distinguish one color from many colors. This lets you build more complex systems and thus help you in the long run. Just keep in mind that a color in VL is something different than a spread of colors.

More about data: [Basic Nodes and Data Types](#) and [More On Data](#)

Data Flow

Data flow in a prototyping environment is about playing with data. For that you place some nodes that offer a promising functionality on your patch which is our name for the canvas. After that you connect the node's data hubs (also known as pins). By that you let data flow. Together with the promising nodes this makes up your program.

In the beginning after placing some nodes you'll encounter a whole lot of possible connections you can make. The connections are called links, since they link data source and data sinks. A node e.g. has inputs for the needed data. Inputs on a node thus are data sinks. Outputs of a node offer the data computed by the node to be used somewhere else. Outputs of a node thus are data sources.

Typically you'll be helped by the programming environment to know which links will make any sense. Throughout the system a data source can only be connected to a data sink (and vice versa). All data hubs also have a currently associated data type. That is: the system knows which data to expect. This will help you to some degree to prevent some errors which would be hard to find otherwise.

Programming may feel like playing. You are in the middle of creating something! To help you within the process of creating, you may force a connection with SPACE-Click, even if the system currently thinks it doesn't make sense. It may make sense a few clicks later after disconnecting another link!

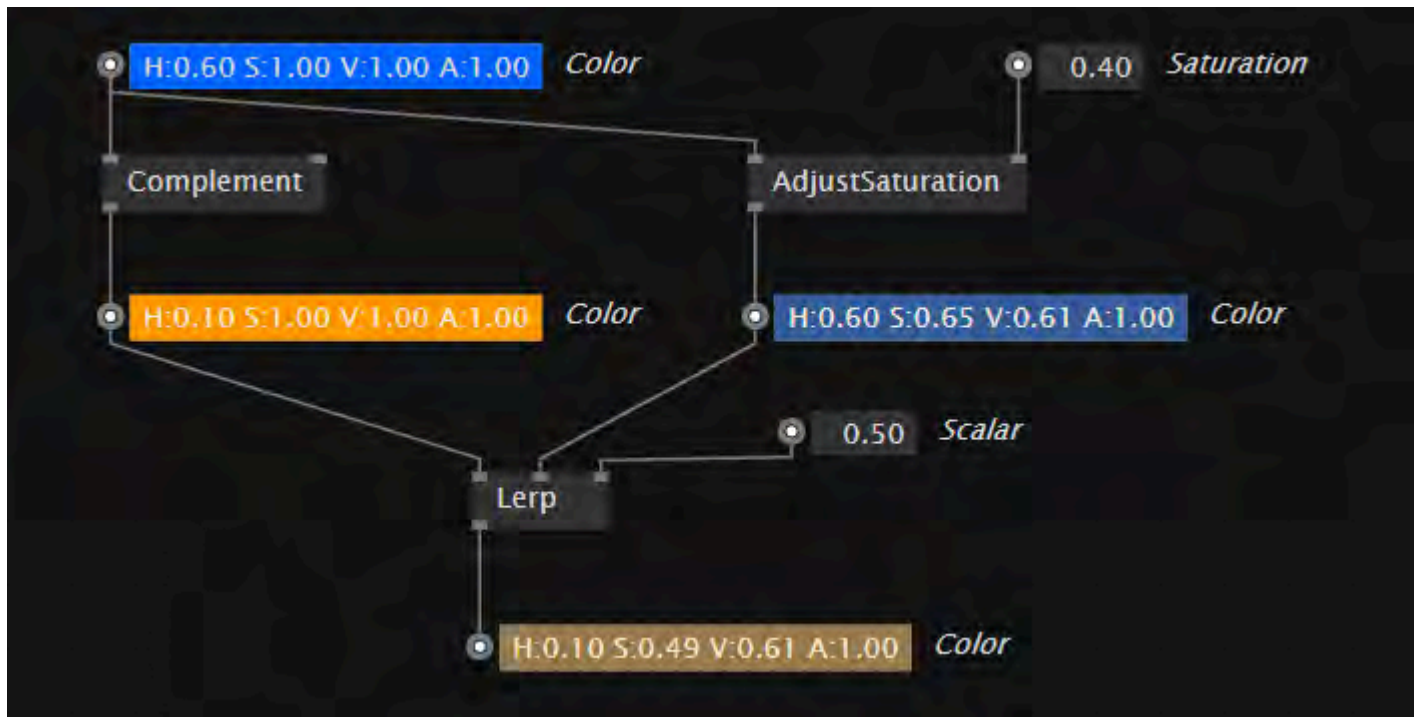
For more on patching experience and work flow see "The Development Environment".

Meaning?

When programming visually with data flow you can link some data from one node to the other. All these nodes are doing their thing and together they form something new, a new functionality.

At times this is fun, at times this is hard work and needs some thinking. But one way or the other: your goal should always be to strive for a patch that not only solves a particular task and is able to be executed by the machine but also is readable for you later on. Practice making notes and explaining the patch to yourself.

For now of course we'll try to do the job of explaining. Here is an example patch:



color-data getting processed with operations

The color constant in the top-left is a data source and holds our data we want to process. As you can see the output of the color is connected to two inputs. On the left side it is connected down to a node which will compute the complementary color and on the right side you see it connected to a node which is used to adjust the saturation. Finally the two data-paths get joined again in the Lerp-node which will give you a color in between the two, positioned by the scalar-input. (0.0 - would give you the first color, 1.0 - would give you the second color)

Defining versus Using

Up to now we looked at what happens inside a patch. We were using nodes, applying nodes, calling nodes. All these terms mean the same thing: usage.

But while patching we built up new functionality. So we should be able to use that somewhere else. And of course this is possible. Like in vvvv, patches define new nodes.

So a patch defines a node? Actually one patch may define a bunch of different nodes as we'll see later on.

So when looking at a patch:

- we see the usage of other nodes
- we see that those nodes may be used within regions

But there is more:

- we see the definition of our own nodes
- we might even see the definition of our own data type.

Because of the fact that we are looking at quite different things at the same time, we'll try to separate both definition and usage throughout the book when not obvious through the context.

We'll have a brief look at nodes in the next chapter, but postpone defining nodes to [Patches as Defining Constructs](#).

One Note for users coming from vvvv:

In vvvv a sub patch behaves like a new instance of the patch. Using the sub patch several times leads to several patch windows docked into each other, right? Right clicking the node representing a sub patch opened up the particular instance. In VL this is different. There is one place to define the functionality, not several synced places for the same patch.

Nodes

As already stated: Nodes are all about using functionality that is already available. When talking about usage we often also use the term application. By using a node you're applying an available functionality in a certain way.

VL offers different flavors of nodes. Let's have a look:

Operation Applications

An operation application applies an operation.

An operation is something very primitive. It is a temporary computation that needs data and yields data. It may have side effects like writing to a file, but typically it is all about computing data based on data.

You feed data to an operation application via its input pins. It computes and hands you the results. Think of it as something volatile and temporal, like a chemical reaction.

The output doesn't depend on previous calls to the operation. It doesn't know of previous calls.

Here are some operation applications:

[TODO: snapshot]

Process Applications

A process application applies a process.

A process is something persistent. It sticks.

Think of the whole program as a process. It sticks around for a while and evolves. Process Applications are modeled by this idea. They can be thought as little programs that stay around, like a living cell.

Here are some process nodes:

[TODO: snapshot]

Compare their look with the operation applications. It is just a small difference that may help you when reasoning about a patch in the future.

All the process applications above need to store a state to function properly. A FlipFlop needs to store its toggle state, a S+H needs to store the last sampled value, a TogEdge needs to store the old input value to be able to detect an UpEdge or a DownEdge in the signal.

The outputs of a process application thus depend on the inputs AND on the state of that process.

more on nodes: [Nodes](#)

The running System

This may sound like cheating, but think of the whole running system as a process node. It gets created once and then updated repeatedly.

A Root Patch

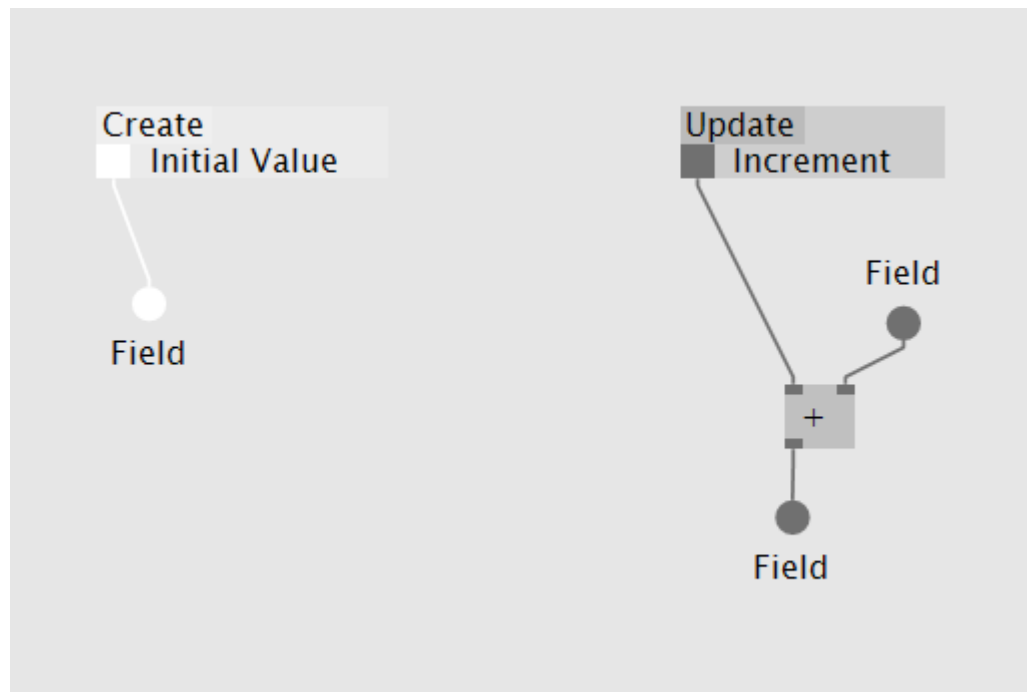
The root patch defines the process.

vvvv user: When cloning a VL template from within vvvv you create a new root patch. The vvvv node contains and manages one instance of the process you just created.

Create And Update

Within the root patch you define what's happening for when the system starts and when the system runs. For that you may place nodes onto "Create" and onto "Update".

As we are still looking at things:



Again: a part of the patch is only executed once when the system starts. The other part is then called over and over again. Can you tell which is which?

vvvv user: when creating or resetting the node from within vvvv the process gets reset: "Create" is called. From then on "Update" is called each vvvv frame.

Persistency

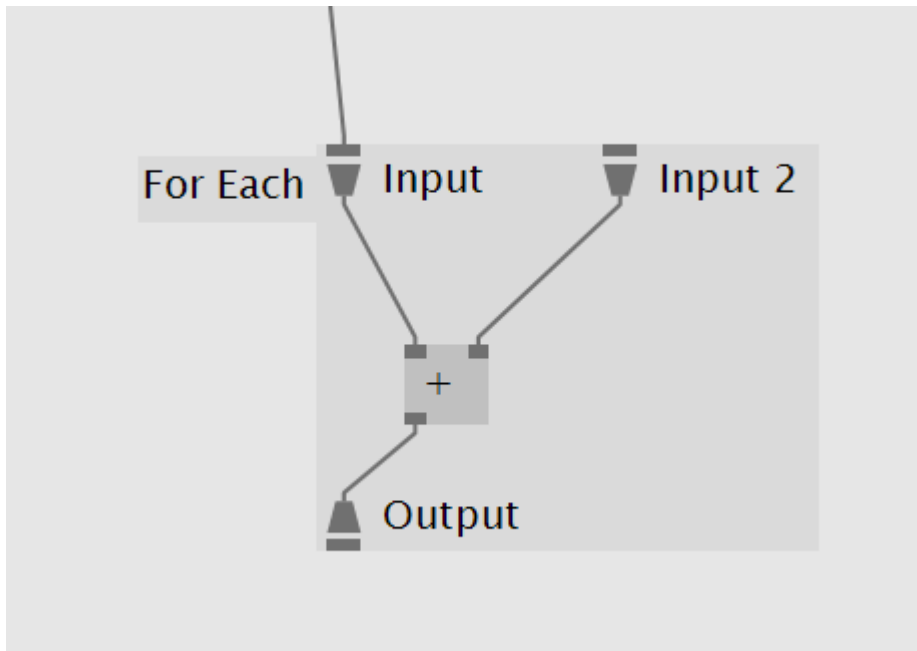
The root patch as already stated is like a process node. It sticks around and may evolve. It may store data internally and retrieve it next frame.

Regions

ForEach

A ForEach region let's you iterate over several collections at the same pace. For each collection you get one item inside the region to work with. You also may output several items per iteration resulting in several spreads of the same length.

The body of the region is called as long as each of the upstream connected sequences still have data. Meaning: If one input has nothing attached to it it won't iterate.



Repeat

A Repeat region let's you define how many times the body should be called. This is done via the Iteration Count Pin.

In all other regards it is very similar to the for each region. It allows to connect several lists and walks over those at the same pace. When a list has fewer items to offer than needed the loop cycles through the list as long as needed.

If

The body of this region is only executed when the condition holds.

You may have zero to many outlets to make nodes outside the region use the data you compute when the region is called. In the case that the condition does not hold the default value is used that may set ontop of the region.

Delegate

With a delegate you may define an operation anonymously. So this new operation then can't be called, since how to name something anonymous? In order to fix that the delegate region has an output that holds the operation defined within the region - as a value.

There are nodes that can call such a delegate if you link that delegate value to them. The most trivial ones are the "Apply" nodes, but there are many others.

Note: All other regions execute right away (well: 0 to many times). The delegate region however does not call the body. It just captures your logic and puts that in a value that can flow over a link. The downstream connected patch is the one that may call this anonymous operation whenever it decides to. You have no influence on that. The point of the delegate is: you don't want to.

Often you start the other way around having a node that needs a delegate that it may call at any time it wants to. It may feel more obvious how to work with delegates when starting in that manner. An example node that needs a delegate is "Where [Spread]". Create it as a node and feed a delegate to it.

[TODO. snapshot]

Operation Region

An operation region combines an operation that needs a delegate with the definition of that delegate.

[TODO. snapshot]

Where

The "Where [Spread]" region allows you to decide for each item if this item shall be included in the resulting spread or not. Taking the perspective of the where node: it calls the body of the region for each item. Each time it does this it hands over an item to the region and uses the your return value to decide if the item should be added to the result.

Categories and Versions

Nodes And Data Types are sorted in a hierarchical category structure.

Categories help you to get an overview of all nodes dealing with a certain aspect, like e.g. mouse input. The node browser allows you to navigate through the categories.

[TODO: snapshot of nodebrowser showing the IO category]

The category of a node is shown in square brackets in the tooltip.

[TODO: tooltip snap]

Also types are categorized. Sometimes you need to specify a type for a data hub. In rare cases you may need to also specify the category if several data types of the same name exist in different categories.

[TODO: type annotation snap]

A version is always shown in round brackets. The main reason for versions is that the author of a node wanted to keep the name as short as possible. Versions help separating the main idea of a node from the details how to use it. There may be several nodes that basically do the same thing but differ in how they do it.

[TODO: snap nodebrowser. different versions]

Documents, Dependencies and Scopes

Documents

A document can have many patches.

In the patches you want to use nodes that either come with the basic library or other nodes that somebody else created.

Dependencies

Thus to be able to use nodes that are not in the basic library you need to be able to tell the system where the file is that holds this other node. We call these files dependencies of the document. You can edit dependencies via the navigation bar.

Dependencies may be other .vl-files or .vlimport-files.

Scopes

Each document sees different nodes depending on its dependencies.

What's good: it doesn't matter if you open several documents at the same time. They don't influence each other as long they don't reference each other. Each document can only see the nodes that can be found in the list of its dependencies.

That way you may open several versions of your document. All the nodes and types that you define in there very likely have the same name in the different versions of your document. But since they don't see each other, both can live besides each other at the same time.

Basic Nodes and Data Types

A glimpse into the most basic nodes and data types the library has to offer.

Numeric Values

Floating Point numbers

A Floating point number is what most people would call a real number or a decimal number, like 12.34.

The reason for calling them floating point is the way they are stored internally which might be a flawed reason for a name. However since there are different ways to encode a real number we stick to calling those by the way they work internally.

And while at it we decided to go one step further than other languages in the cryptic naming of our number types. We add bit count to their name.

- Float32
- Float64

The reason for that is that both 32bit floating point and 64bit floating point are widely used. In the long run every language that wants to be precise on what the machine must do needs to be able to work with both data types. And there are more: 16bit floating points, 128bit floating point. You get the point. Using another name for each feels wrong. The only thing that separates these data types is the precision with which they manage to store a real number value.

Integers

Surprisingly with integers there are even more different types. All just for storing whole numbers. They start with 8 bit and typically go up to 64 bit. There are even unsigned and signed versions. Signed integers are able to store negative and positive numbers. Unsigned versions can only store positive numbers. In the long run we'll have them all. We are starting with the signed versions that can hold negative numbers also:

- Integer32
- Integer64

Where any other language would need to explain the names and what precision they stand for I guess I don't need to elaborate on this given their names.

Boolean

A boolean value is either 0 or 1 a.k.a. false and true.

Compatibility

A boolean is either 0 or 1. But these are whole numbers also! So booleans are integers? Yes. When a data sink needs an integer value, you may link boolean data to it.

A number 5 is an integer but it also is a real number. Whole numbers are part of all the real numbers. Real numbers are not defined to be fractional, they can be just whole. It's fine. Because of this you may feed integer data into a float data sink.

In both cases an implicit conversion is happening.

Text

Char

A character is modeled with the Char data type. 'A' is data of type Char.

String

A text is modelled by a chain of characters that is called a String. "Hello World!" is of type String.

Spreads and Other Collections

Spread

Spread is a name that we stole from vvvv. It is just a collection of items that is ready to be consumed. In other words: There is some memory in the computer that holds all the items. All the info is there. They can be read, the count is known, their values are known.

What's more though: A spread can't mutate. A spread coming from one data source can be fed to many data sinks and we can guarantee you that all data sinks will always access the same data.

Confusion #1:

Of course the data source of type Spread may change its output over time. But it will do so by retrieving new spreads.

Confusion #2:

Of course there are nodes to "change" the content of a spread. You may want to write a new value into it, delete something or add some new item. But again these operations will create new spreads. They won't change the spread coming from upstream.

Here are some operations the spread offers. Some of them are offered as Operation Regions, most of them are nodes.

[TODO: snapshot]

SpreadBuilder

There is something called SpreadBuilder that offers a more efficient way to create spreads. A spread builder is mutable and should only be used in places where you need to do many sequential changes on a spread. If you are sure you are ready for performance optimizations use the ToBuilder node to convert a spread to a builder. And when done convert it back to a regular spread with the ToSpread node.

We would like to encourage you to use spread builders only locally: to create spreads. Pass around the spread that you just built. Don't pass the builder itself. Even when you need to store a spread for later usage: store the spread, not the builder. It helps when reasoning about a patch.

HashSet

The hash set models a (mathematical) set of values.

If you have a set of integers, you may have the number 2 in there. Once. You can't have it there more than once. The idea of a set is that it holds elements of a certain type. Each possible value of that data type is either in the set or not.

Sequence

All the data types above can be seen as sequences of data.

There is a way of looking through their content item by item.

A "ForEach" loop region e.g. needs just that to be able to make the incoming items available in the body of the region.

But from the start: there are different ways of storing items in different collection types. They differ in details that sometimes do not matter! And **one aspect** of all those collections is that you may look through them item by item.

That is what the sequence type is for. To offer a common ground. If a data sink only needs to sequentially look through all items it may accept a sequence. If you create a node for others to use you might also just accept a sequence. The user of your node which might be yourself might be happy to be able to feed any collection he/she wants to.

List

The Repeat Region accepts lists. A list is also an abstraction, like a sequence is. It is any collection that has a count.

Spread, SpreadBuilder and HashSet are all sequences and lists.

More On Data

Data and Mutability

There are basically two different ways to think about data. You might state: "Data is immutable. Red will be always red. And 15 will always be 15. The number 15 doesn't mutate, age or transform. Neither does Pi." But you may also think about data as something that is stored in some memory of the computer. Isn't this also data? Something that can be stored in memory? Of course. So what if the value changes that is stored at that particular place in memory. Didn't the data itself change?

We won't solve the issue of confusion. We can just try to be as precise as possible and by separating both ideas. So what we do is: we actually treat both as different data. A color for us will always be immutable. If you need the notion of "a mutating color" we may later on look into ways of letting you do this, but the main point is that you would have to model a new data type to do this. Colors themselves are immutable. Discussion over.

By doing this, we are separating the differing ideas of what data is again with the same approach that helped us to separate all the possible data earlier on: via data types. Mutability or immutability is just something that can be said about a type:

A color is immutable. A number is immutable. A string (text) is immutable. And your own data types will also be immutable by default.

Immutability is our default

When talking about data later on in the book you should always think of that data as being immutable. Most of our basic data types are immutable. As soon as we encounter mutable types we'll also use the term reference (to some location in memory) or (shared) memory to underline the difference to the default.

While other systems embrace the idea of data being a reference to mutating memory mainly out of low level performance reasons, we embrace the idea of immutable data and encourage you to deal with this kind of data as long as possible.

Immutable data makes reasoning about data flow much more pleasant than when dealing with references. We'll look into data flow in the next chapter. Immutable data also has advantages when reasoning about parts of a program running in parallel on different cores of a modern processor, which is another reason for our decision of making immutable data the default.

DataFlow and Mutability

TODO: place the same image from data flow in here.

Immutable Data and Data flow

Let's focus on the HSV nodes' output pin which is a data source for two other nodes. Two links start from here, so both receiving nodes will receive that same color computed by the HSV node. Remember: only the data source should be the source of data. There is no way of messing this up. And there should be no way of messing this up.

This is essential to be able to understand the patch. The data is coming from up there, solely from the data source (the output pin of the HSV node). That's all that matters. Any exception to that rule would destroy the basic idea of data flow that is **data is flowing from source to sink only**.

Mutable Data and Reference flow

Mutable Data also flows downwards, however this time it is more meaningful to state: a reference to the data is flowing downstream. The receiving nodes now can change the data in-place, meaning that every other sink that got handed a reference to that object will see the changed object.

The fact that sinks can influence each other, makes it more important to not only think about data flow, but also execution order. You will need to think more imperatively, less in a declarative way. In particular you will try to patch more vertically, where the patch reads more like a big list of instructions, that only can be executed in a certain way.

Detecting Changes

Recap of immutable data and how to produce new values

They are easy to digest in dataflow oriented patches, as you never need to reason about when you look at something immutable, it's always in that frozen state. The only way to have just a little bit of joy with not changing memory is to allocate some new memory, produce new objects, which capture the new immutable value. So, when changing a property in a Record, you get a new snapshot, a new object that captures that new state. When adding a slice to a spread you get a new spread object.

Is the data flowing over a link changing over time?

The "Changed" node should probably be called ChangedReference as it is only capable of detecting exactly those changes where it gets confronted with new objects. It can't detect changes inside mutating objects. This however is also interesting sometimes. E.g. in Elementa or bigger object graphs, you want to mutate your objects in order to avoid huge memory allocations all the time.

Mutable Object "Snapshots"

The idea of having different "snapshots of object over time" is just very nice. If there is an easy way to detect that last time you saw snapshot 17 and now it's snapshot 31 of that same object then you'd even be noticing changes that happened a while ago when you weren't watching. I like to call this a ticketing system, and I think it's also called that way in Elementa. As a lib developer, you just add an integer property "Ticket" to your object and count up whenever you want to do changes to a mutating object. Now you can check whether a Ticket of an object has changed: with the Changed node.

Immutable Object Snapshots share an Identity

Btw. another interesting fact regarding snapshots of Records: when producing a new snapshot by adding a slice to a spread or setting a property of a Record you get a new object, that shares the "Identity" with the object that it was "cloned from". Thus: you can "see" that certain snapshots belong together.

Changed nodes are everywhere

In the tooltip the "clock" also visualizes whether the data changed. Filled means changed object, stroke-only means same object as last frame.

Also, each input border controlpoint of a Cache region contains a Changed node.

Builders of Immutable Data

Immutable types - like spread - are easy to digest in dataflow oriented patches, as you never need to reason about when you look at something immutable, it's always in that frozen state. The only way to have just a little bit of joy with not changing memory is to allocate some new memory, produce new objects, which capture the new immutable value. So, when changing a property in a Record, you get a new snapshot, a new object that captures that new state. Now imagine the Spread being such an immutable thing. When inserting or deleting (...) slices you often enough produce a complete new collection. That's why it's sometimes helpful to use a builder for that immutable thing. A builder captures the idea, that while trying to describe the spread you want to have, you are completely fine with this thing being mutable - thus not producing any temporary collections you don't need. When done, basically when letting it flow out of your patch in order to be consumed by other patches, the idea would be to turn it into something easily digestible again: a spread.

Note: There are different clever builders. E.g. the Cons node also uses a builder, but one that is aware which values got consed last frame, and if those are the same as this frame, then no new spread gets produced. Similar the builder that is used inside a splicer output of those loop regions.

But ok, that's yet another topic: "managing snapshots of spreads over time, producing as few garbage as possible".

While starting from some spread, switching to builder, doing some modifications and switching back to spread is a more established concept that works in operations as well and doesn't have anything to do with comparing spreads over time...

Generics

Sub Types

As already mentioned in [Spreads and Other Collections](#) a Spread can be seen as a List or just a Sequence.

Some spread (0, 1, 2) has the type Spread. Nobody wants to discuss that.

And still this data (0, 1, 2) can also be understood as data of a more general purpose type. We can even state that a Spread is just some object. This is not saying much but still is true. At some point we might want to put different things into one Spread. This is possible with a Spread of object, since anything is also an object.

Up to now there are only built in sub type relationships. Later on you might be able to design your own abstract super types and patch your own sub types that belong to the super type.