Using JUCE ValueTrees and Modern C++ to Build Large Scale Applications

David Rowland
Tracktion

Notes

- Slides are numbered
- Slides/examples will be on GitHub
- Code style tweaked for slide presentation

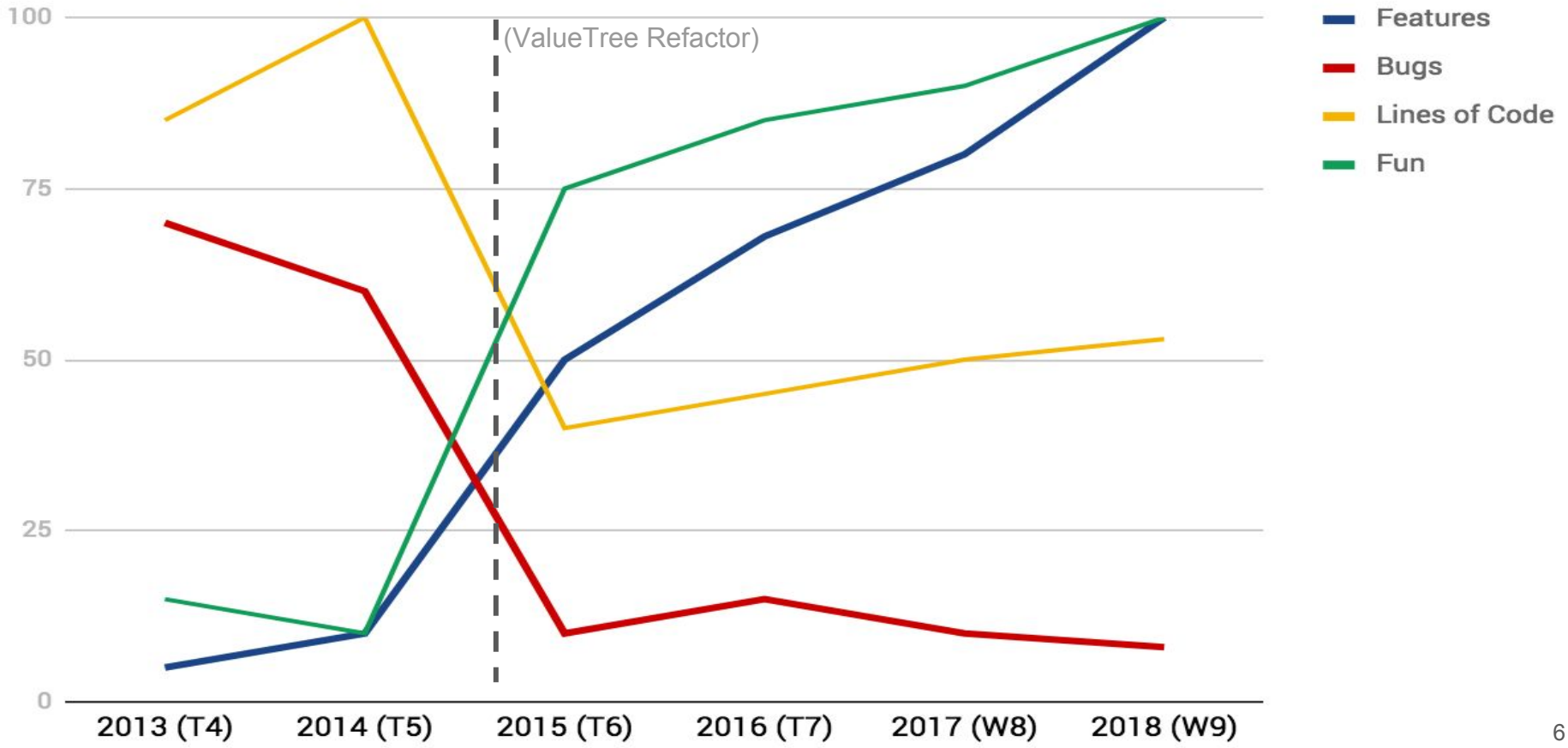
1. Introduction

What's it all about? Waveform

Why ValueTrees?

- 4 years ago restructured Tracktion to use ValueTrees
- Refactored ½ million line code base
- Now ~¼ million lines

Tracktion Development Over Time



Intro

- Overview of `ValueTree` related classes
  - `var`, `Identifier`, `ValueTree`, `Value`
- In-depth look at `ValueTree`
  - Callbacks, lifetime/reference counting, thread safety
  - Custom data storage, serialisation, undo/redo
- Using `ValueTrees` as application data models
- Building type-safe object lists

2. Explanation of var, ValueTree, and Value classes

juce::var

- Variant type
- Analogous to a Javascript var
- Can be used to store:
  - Primitive types (int, int64, bool, double)
  - juce types (String, MemoryBlock)
  - juce::Arrays of vars Array<var>
  - juce::ReferenceCountedObjects
  - Callable juce::DynamicObjects
- Can be serialised to JSON
- Will convert between each other using appropriate operators

```
var v (3.141);
DBG("Type: " << getVarType (v));
DBG("Value: " << static_cast<double> (v));
DBG("Convert to a String: " << v.toString());
DBG("Type: " << getVarType (v)); // Still a double

v = "Hello World!";
DBG("Type: " << getVarType (v));
```

</> Output:

```
Type: double
Value: 3.141
Convert to a String: 3.1410000000000000142
Type: double
Type: String
```

juce::Identifier

- juce::Identifier is like an enum
- Actually a globally pooled String
- Comparisons between Identifiers are pointer comparisons so extremely quick
- Creating an Identifier can be slow (O(log n) where n is the number of existing Strings in the pool)
- The best way to create them is statically so they are created at app startup
- 🔔 Use a global set of Identifiers and a macro to create (use the same name as the ID for readability and avoid spelling mistakes)

```
namespace IDs
{
    #define DECLARE_ID(name) const juce::Identifier name (#name);

    DECLARE_ID (TREE)
    DECLARE_ID (pi)

    #undef DECLARE_ID
}

DBG(IDs::TREE.toString());
DBG(IDs::pi.toString());

const Identifier localTree ("TREE");
const String treeString ("TREE");
const Identifier treeFromString (treeString);

if (IDs::TREE == localTree && localTree == treeFromString)
    DBG("All equivalent");
else
    DBG("Not the same");
```

</> Output:

```
TREE
pi
All equivalent
```

juce::ValueTree - Introduction

- juce::ValueTree is analogous to XML
- Its structure is the same (without text elements)
- Imagine if you could have an XmlElement but listen to when attributes are changed and children are added/removed/moved. This is juce::ValueTree!
- Terminology:
  - attribute -> property
  - tag -> type

```
ValueTree v (IDs::TREE);
DBG(v.toXmlString());

ValueTree clip (IDs::CLIP);
clip.setProperty (IDs::start, 42.0, nullptr);
clip.setProperty (IDs::length, 10.0, nullptr);
v.addChild (clip, -1, nullptr);
DBG(v.toXmlString());
```

</> Output:

```
<TREE/>

<TREE>
  <CLIP start="42" length="10"/>
</TREE>
```

juce::ValueTree - Reference Counting

- ValueTrees are actually lightweight wrappers around an internal reference counted object called a SharedObject
- Copying a ValueTree simply increments the reference count of the SharedObject
- This means it's cheap to store and copy ValueTrees as they point to shared data
- You can create a unique copy using `ValueTree::createCopy`

```
ValueTree v1 (IDs::TREE);
DBG ("1. Ref count v1: " << v1.getReferenceCount());

ValueTree v2 (v1);
DBG ("2. Ref count v1: " << v1.getReferenceCount());
DBG ("3. Ref count v2: " << v2.getReferenceCount());

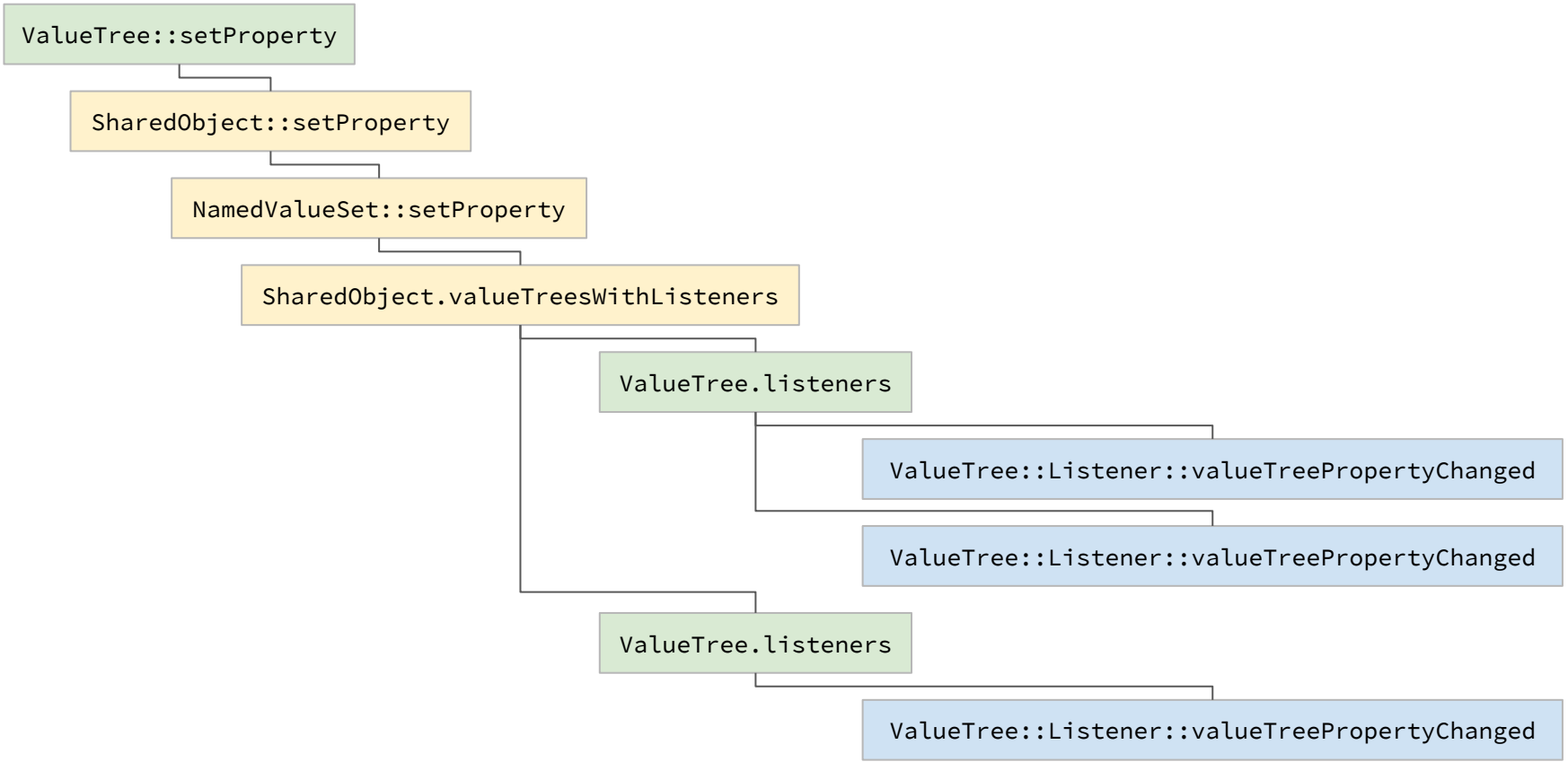

ValueTree v3 (v1.createCopy());
DBG("4. Ref count v3: " << v3.getReferenceCount());
```

```
</> Outputs:
1. Ref count v1: 1
2. Ref count v1: 2
3. Ref count v2: 2
4. Ref count v3: 1
```

juce::ValueTree - Callbacks (1)

- ValueTrees have callbacks when:
  - Properties change
  - Children are added/removed/moved
  - The parent changes
- These are essentially passed up from changes to the internal SharedObject
- Additionally there is a callback to be notified when a ValueTree is re-assigned i.e. the SharedObject is changed
- ✎ Callbacks are synchronous

juce::ValueTree - Callbacks (2)



juce::ValueTree - Callbacks (3)

- When you register as a listener, the **ValueTree** holds a pointer to the listener, not the SharedObject
- 🔔 This means it's usually best to take a copy of the ValueTree and register with that
- ✎ Callbacks iterate upwards in the tree hierarchy so listeners to parent nodes will receive property change callbacks for deeply nested trees
- Make sure to design your tree structure accordingly and check types/properties in each of your callbacks

```
struct Widget   : public ValueTree::Listener
{
    Widget (ValueTree v) : state (v)
    {
        state.addListener (this);
    }

    ValueTree state;

    void valueTreePropertyChanged (ValueTree&, const Identifier&) override {}
    void valueTreeChildAdded (ValueTree&, ValueTree&) override {}
    void valueTreeChildRemoved (ValueTree&, ValueTree&, int) override {}
    void valueTreeChildOrderChanged (ValueTree&, int, int) override {}
    void valueTreeParentChanged (ValueTree&) override {}
};
```

juce::ValueTree - Callbacks (4)

- ✎ Don’t rely on the order of callbacks i.e. don’t assume one callback will have happened before another for the same property
- Use an AsyncUpdater to ensure all callbacks have happened and your concrete objects have the correct state before using them

juce::ValueTree - Undo/redo

- Because all changes to a ValueTree are either property changes, child add/remove or child moved actions, they can very easily be undone/redone simply by supplying a juce::UndoManager to the actions
- If all your ‘views’ simply respond to the state of the tree and are updated when it changes, everything will always stay in sync

```
UndoManager um;
ValueTree v (IDs::CLIP);
v.setProperty (IDs::start, 0.0, nullptr);
v.setProperty (IDs::length, 42.0, nullptr);
DBG (v.toXmlString());

um.beginNewTransaction();
v.setProperty (IDs::length, 10.0, nullptr);
DBG (v.toXmlString());

um.undo();
DBG("Undoing:");
DBG (v.toXmlString());
```

```
</> Output:
<CLIP start="0" length="42"/>
<CLIP start="0" length="10"/>
Undoing:
<CLIP start="0" length="42"/>
```

juce::Value (1)

- juce::Value has similarities to juce::ValueTree but wraps a single juce::var
- Internally it has a ValueSource which by default acts as a reference counted var
- You can listen for changes to the single var
- ✎Value callbacks are asynchronous*
- Useful for attaching to UI

```
Value v1 (42);
Value v2 (3.141);

DBG("v1: " << v1.toString());
DBG("v2: " << v2.toString());

v2.referTo (v1);
DBG("v2: " << v2.toString());
```

```
</> Output:
v1: 42
v2: 3.1410000000000000142
v2: 42
```

juce::Value (2)

- You can also create a custom ValueSource which you can either use to get synchronous change messages
- Or wrap custom data as a var
- You can create Values from ValueTree properties
- This creates a custom ValueSource that is a ValueTree::Listener

```
struct SyncronousValueSource    : public Value::ValueSource
{
    SyncronousValueSource() = default;

    SyncronousValueSource (const var& initialValue)
        : value (initialValue)
    {
    }

    var getValue() const override
    {
        return value;
    }

    void setValue (const var& newValue) override
    {
        if (! newValue.equalsWithSameType (value))
        {
            value = newValue;
            sendChangeMessage (true);
        }
    }

private:
    var value;
};
```

juce::Value (3)

```
struct Transport : public ChangeBroadcaster
{
    Transport() = default;
    void start()    { isPlaying = true; sendSynchronousChangeMessage(); }
    void stop()     { isPlaying = false; sendSynchronousChangeMessage(); }

    bool isPlaying = false;
};

struct TransportValueSource : public Value::ValueSource,
                              private ChangeListener
{
    TransportValueSource (Transport& t) : transport (t) {}

    var getValue() const override
    {
        return transport.isPlaying;
    }

    void setValue (const var& newValue) override
    {
        if (newValue)   transport.start();
        else            transport.stop();
    }

    void changeListenerCallback (ChangeBroadcaster*) override
    {
        sendChangeMessage (true);
    }

    Transport& transport;
};
```

```
Transport transport;
Value transportValue (Value (new TransportValueSource
(transport)));
DBG("playing: " << (int) transport.isPlaying);
DBG("value: " << (int) transportValue.getValue());

DBG("\nSTARTING");
transport.start();
DBG("playing: " << (int) transport.isPlaying);
DBG("value: " << (int) transportValue.getValue());

DBG("\nSETTING VALUE: 0");
transportValue.setValue (false);
DBG("playing: " << (int) transport.isPlaying);
DBG("value: " << (int) transportValue.getValue());
```

```
playing: 0
value: 0

STARTING
playing: 1
value: 1

SETTING VALUE: 0
playing: 0
value: 0
```

```
struct Transport : public ChangeBroadcaster
{
    Transport() = default;
    void start()    { isPlaying = true; sendSynchronousChangeMessage(); }
    void stop()     { isPlaying = false; sendSynchronousChangeMessage(); }

    bool isPlaying = false;
};

struct TransportValueSource : public Value::ValueSource,
                              private ChangeListener
{
    TransportValueSource (Transport& t) : transport (t)
    {
        transport.addChangeListener (this);
    }

    ~TransportValueSource()
    {
        transport.removeChangeListener (this);
    }

    var getValue() const override
    {
        return transport.isPlaying;
    }

    void setValue (const var& newValue) override
    {
        if (newValue)   transport.start();
        else            transport.stop();
    }

    void changeListenerCallback (ChangeBroadcaster*) override
    {
        sendChangeMessage (true);
    }

    Transport& transport;
};
```

```
Transport transport;
Value transportValue (Value (new TransportValueSource (transport)));
DBG("playing: " << (int) transport.isPlaying);
DBG("value: " << (int) transportValue.getValue());

DBG("\nSTARTING");
transport.start();
DBG("playing: " << (int) transport.isPlaying);
DBG("value: " << (int) transportValue.getValue());

DBG("\nSETTING VALUE: 0");
transportValue.setValue (false);
DBG("playing: " << (int) transport.isPlaying);
DBG("value: " << (int) transportValue.getValue());
```

</> Output:

```
playing: 0
value: 0

STARTING
playing: 1
value: 1

SETTING VALUE: 0
playing: 0
value: 0
```

Custom Data Storage

- You can store custom data by converting to/from Strings or MemoryBlocks
- Since it may be slow to do these conversions you might want to use a CachedValue
- By specialising var conversions, you only do the String conversion when the data actually changes
- Some data is external to your model and will require flushing to the tree before serialisation e.g. AudioProcessor state

```
template<>
struct VariantConverter<Image>
{
    static Image fromVar (const var& v)
    {
        if (auto* mb = v.getBinaryData())
            return ImageFileFormat::loadFrom (mb->getData(), mb->getSize());

        return {};
    }

    static var toVar (const Image& i)
    {
        MemoryBlock mb;

        {
            MemoryOutputStream os (mb, false);

            if (! JPEGImageFormat().writeImageToStream (i, os))
                return {};
        }

        return std::move (mb);
    }
};
```

Use CachedValue<> to Wrap Properties as Objects

- Allows “property” type get/set methods
- Simple primitives are easy:

```
CachedValue<float> start;
start.referTo (v, IDs::start, um);
```

- Technically not thread safe...

Use CachedValue<> to Wrap Properties as Objects (2)

- More complex objects can be stored as `var` wrapped `Strings` or `MemoryBlocks`
- Specialise `VarientConverter<>` to do this transparently

```
CachedValue<Image> image;
image.referTo (v, IDs::image, um);
image = Image::loadFrom (imageFile);
```

```
template<>
struct VariantConverter<Image>
{
    static Image fromVar (const var& v)
    {
        if (auto* mb = v.getBinaryData())
            return ImageFileFormat::loadFrom (mb->getData(), mb->getSize());

        return {};
    }

    static var toVar (const Image& i)
    {
        MemoryBlock mb;

        {
            MemoryOutputStream os (mb, false);

            if (! JPEGImageFormat().writeImageToStream (i, os))
                return {};
        }

        return std::move (mb);
    }
};
```

Serialisation

- juce::ValueTree can be easily serialised to either XML, a binary format or a compressed binary format
  - XML is plain text so can be debugged
  - Slightly slower than binary however as it will have to convert to a juce::XmlElement first
  - Type information is lost (will be reloaded as Strings)
- Binary will be fastest to read/write and var types are correctly saved/recalled because the juce::var stream operations are used which write type markers
- Compressed binary is same as binary format but can take up less space
- Slower to read/write

```
ValueTree v (IDs::TREE);
v.setProperty (IDs::pi, double_Pi, nullptr);
DBG("Type before: " << getVarType (v[IDs::pi]));

std::unique_ptr<XmlElement> xml (v.createXml());
DBG("Type after XML: " << getVarType (ValueTree::fromXml
(*xml)[IDs::pi]));

MemoryBlock memory;

{
    MemoryOutputStream mos (memory, false);
    v.writeToStream (mos);
}

MemoryInputStream mis (memory, false);
DBG("Type after binary: " << getVarType
(ValueTree::readFromStream (mis)[IDs::pi]));

</> Output:
Type before: double
Type after XML: String
Type after binary: double
```

Thread Safety (1)

- juce::var is not thread-safe
- juce::Value is not thread-safe
- juce::ValueTree is not thread-safe
- juce::CachedValue<> is not thread-safe*
- *(Data race on a POD type which could be argued is a benign data-race)*
- *Which is still bad!*
- Make sure all ValueTree interactions happen on the message thread! Otherwise your callbacks will happen from the calling thread and cause havoc

Thread Safety (2)

- Use message posting, AsyncUpdater etc. to enforce this
- This should ensure all callbacks happen on the message thread
- Take copies of the data and make that thread safe either with atomics or locks
- CachedValue<AtomicWrapper<Type>>

ValueTrees - Top 5 Things to Know

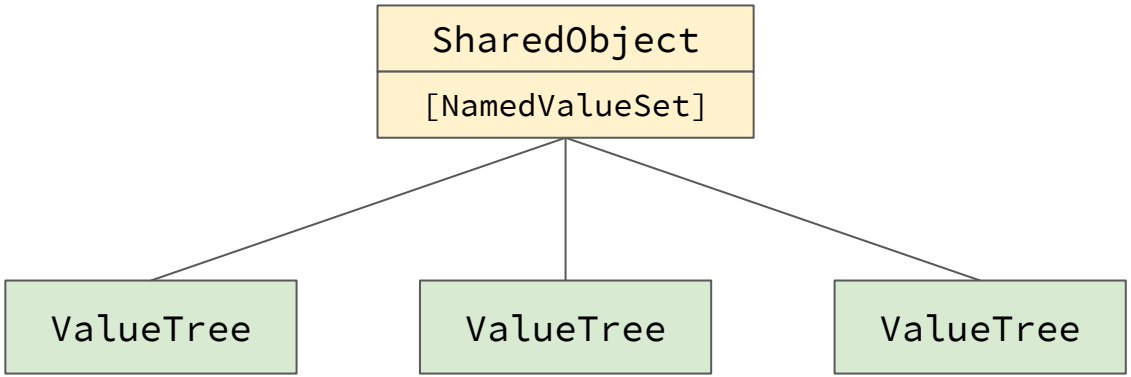
1. ValueTree are Like XML

- `juce::ValueTree` is analogous to XML
- Its structure is the same (without text elements)
- Imagine if you could have an `XmlElement` but listen to when attributes are changed and children are added/removed/moved. This is `juce::ValueTree`!
- Terminology:
  - attribute -> property
  - tag -> type

```
<EDIT appVersion="Waveform 8.1.8" mediaId="e31b7/1d17644"
      creationTime="1402484148538" fps="24" timecodeFormat="beats">
    <VIEWSTATE viewleft="-1" viewright="194.9998083000000122"/>
    <TRACK mediaId="e2b7f/23eff3ab" name="melody"
height="34.214285714285715301"
         colour="ffff4d4c">
        <CLIP mediaId="e31b7/d45f454" name="xxxx" type="midi"
source="e2b7f/23eff3c4"
          start="59.076864000000000487"
length="29.076893999999999352"
          offset="0" colour="ffffab00"/>
        <CLIP mediaId="e31b7/30291232" name="melodyoutro"
type="midi" source="e2b7f/23eff3c4"
          start="147.69216000000000122"
length="29.538432000000000244"
          offset="0" colour="ffffab00"/>
    </TRACK>
</EDIT>
```

2. ValueTrees are Wrappers around a SharedObject

- `ValueTrees` are lightweight objects that manage a reference-counted `SharedObject`
- The `SharedObject` holds a `NamedValueSet` of the properties
- `ValueTree` provides an interface to the `SharedObject`
- It is cheap to copy around `ValueTrees`



3. Listeners are Stored with the ValueTree

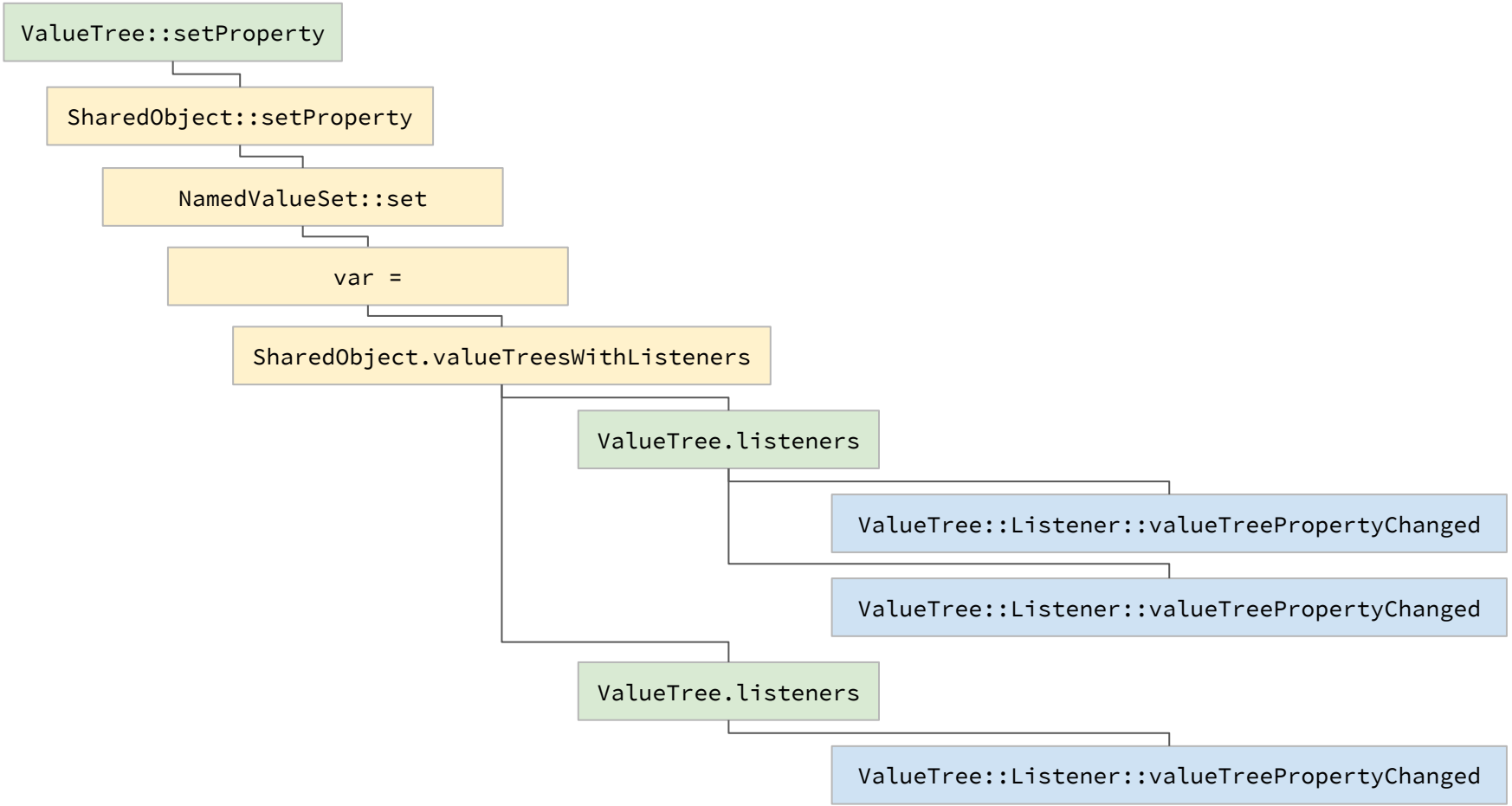
- When you register as a listener, the **`ValueTree`** holds a pointer to the listener, not the `SharedObject`
- This means it's usually best to take a copy of the `ValueTree` and register with that

```
struct Widget   : public ValueTree::Listener
{
    Widget (ValueTree v)
        : state (v)
    {
        state.addListener (this);
    }

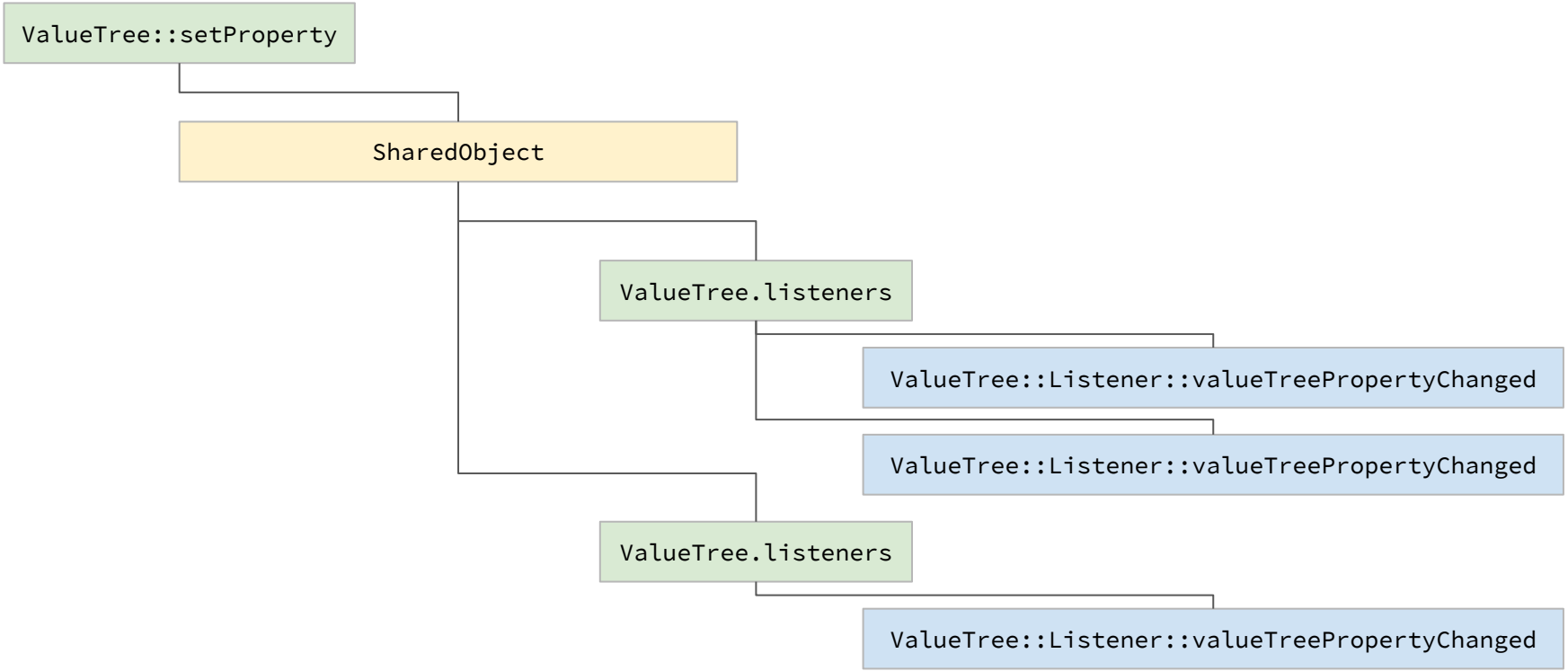
    ValueTree state;

    void valueTreePropertyChanged (ValueTree&, const Identifier&) override {}
    void valueTreeChildAdded (ValueTree&, ValueTree&) override {}
    void valueTreeChildRemoved (ValueTree&, ValueTree&, int) override {}
    void valueTreeChildOrderChanged (ValueTree&, int, int) override {}
    void valueTreeParentChanged (ValueTree&) override {}
};
```

3. Listeners are Stored with the ValueTree (2)



3. Listeners are Stored with the ValueTree (3)



4. Callbacks are Synchronous and Called for all Children

- Callbacks are synchronous so don't do anything time consuming
- Use `AsyncUpdater` to combine actions from property changes

- Callbacks iterate upwards in the tree hierarchy so listeners to parent nodes will receive property change callbacks for deeply nested trees
- Make sure to design your tree structure accordingly and check types/properties in each of your callbacks

```
struct Widget   : public ValueTree::Listener,
                  private AsyncUpdater
{
    Widget (ValueTree v) : state (v)
    {
        state.addListener (this);
    }

    ValueTree state;

    void handleAsyncUpdate() override
    {
        // Sort notes
    }

    void valueTreePropertyChanged (ValueTree& v,
                                   const Identifier& id) override
    {
        if (v.hasType (IDs::NOTE) && id == IDs::start)
            triggerAsyncUpdate();
    }

    //...
};
```

5. Serialise Non-primitive Data to Strings

- Combine multiple, related properties into a single property
- Serialise to/from a `String`
- Think about efficiency here, most properties won't be changing very often. For ones that do, you may need a slightly different paradigm

```
TimestretchOptions options;
options.syncroniseTimePitch = true;
options.preserveFormants = true;
options.envelopeOrder = 128;

ValueTree v (IDs::CLIP);
v.setProperty (IDs::timestretchOptions, options.toString(), nullptr);
DBG(TimestretchOptions (v[IDs::timestretchOptions].toString()).toString());
```

```
</> Output:
0|1|1|128
```

```
struct TimestretchOptions
{
    TimestretchOptions() = default;
    TimestretchOptions (const String& s)
    {
        auto tokens = StringArray::fromTokens (s, "|", "");
        stereoMS            = tokens[0].getIntValue() != 0;
        syncroniseTimePitch = tokens[1].getIntValue() != 0;
        preserveFormants    = tokens[2].getIntValue() != 0;
        envelopeOrder       = tokens[3].getIntValue();
    }

    String toString() const
    {
        StringArray s;
        s.add (stereoMS ? "1" : "0");
        s.add (syncroniseTimePitch ? "1" : "0");
        s.add (preserveFormants ? "1" : "0");
        s.add (String (envelopeOrder));
        return s.joinIntoString ("|");
    }

    bool stereoMS = false, syncroniseTimePitch = false,
         preserveFormants = false;
    int envelopeOrder = 64;
};
```

3. Using ValueTrees as the model of an application

Trees as classes

- `ValueTrees` naturally fit the hierarchical nature of a lot of apps
- Tree types are analogous to class names
- Properties are analogous to member variables

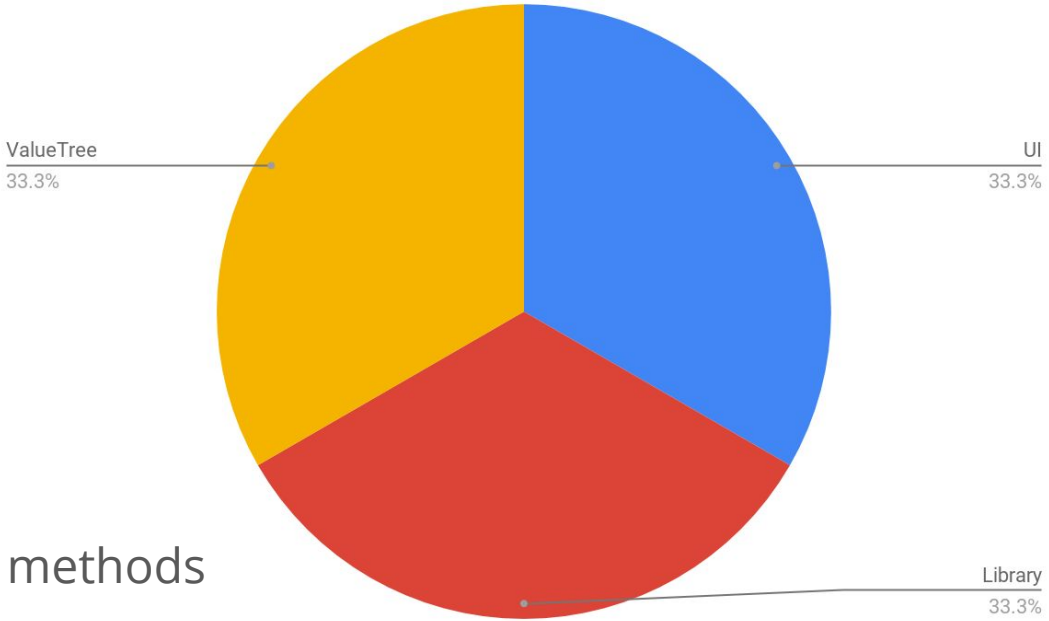
```
<EDIT appVersion="Waveform 8.1.8" mediaId="e31b7/1d17644"
      creationTime="1402484148538" fps="24" timecodeFormat="beats">
    <VIEWSTATE viewleft="-1" viewright="194.9998083000000122"/>
    <TRACK mediaId="e2b7f/23eff3ab" name="melody"
height="34.214285714285715301"
         colour="ffff4d4c">
        <CLIP mediaId="e31b7/d45f454" name="xxxx" type="midi"
source="e2b7f/23eff3c4"
          start="59.076864000000000487"
length="29.076893999999999352"
          offset="0" colour="ffffab00"/>
        <CLIP mediaId="e31b7/30291232" name="melodyoutro"
type="midi" source="e2b7f/23eff3c4"
          start="147.69216000000000122"
length="29.538432000000000244"
          offset="0" colour="ffffab00"/>
    </TRACK>
</EDIT>
```

Trees and MVC

- Using the callback mechanism of `ValueTree` you can build objects
- In an MVC type paradigm, the `ValueTree` takes care of the data model and controller aspects
- You can then write multiple ‘views’ to represent this data model letting the callbacks notify you when things have changed
- Making changes to the tree results in the various views updating themselves to reflect the new state of the tree
- You can have multiple ‘views’ if required

LIVE DEMO - 1_ValueTreeDemo

- ~750 lines of code
  - Tree view of data
  - Selected property panel
  - Colour picker
  - Drag and drop
  - Undo/redo
  - Serialisation
- ~250 UI code
  - Drawing
  - Property panel
- ~250 reusable library/helper methods
  - Saving ValueTree to/from a file
  - juce::Component helpers
- ~250 of ValueTree interaction
  - Creation
  - Validation
  - Get/set properties



Undo/redo

- When mutating a `ValueTree`, simply supply an `UndoManager` to make that action undoable
- `v.setProperty (IDs::start, 42.0, &undoManager);`
- ```
  ValueTree clip (IDs::CLIP);
  v.addChild (clip, -1, &undoManager);
  ```
  ```

4. Scaling Usage to Large Apps

Typed Objects

- Interacting directly with a `ValueTree` is quick but there are pitfalls:
  - No type checking
  - No verification/bounds checking
  - Requires hidden knowledge about the data properties and tree structure
- We like:
  - Abstractions
  - Type-safety
  - Efficiency
- You can build wrappers around `ValueTrees` to enforce all of this

```
struct Clip
{
    Clip (const ValueTree& v) : state (v)
    {
        jassert (v.hasType (IDs::CLIP));
    }

    double getStart() const
    {
        return state[IDs::start];
    }

    void setStart (double time)
    {
        jassert (time >= 0.0);
        state.setProperty (IDs::start, jmax (0.0, time), nullptr);
    }

    Colour getColour() const
    {
        return Colour::fromString (state[IDs::colour].toString());
    }

    void setColour (Colour c)
    {
        state.setProperty (IDs::colour, c.toString(), nullptr);
    }

    ValueTree state;
};
```

```
ValueTree clipState (IDs::CLIP);
Clip c (clipState);
c.setStart (-1.0);          // Triggers assertion (or throws exception)
c.setColour (Colours::red); // Has to be a colour
DBG(c.state.toXmlString()); // Start time is correctly 0.0

</> Output:
<CLIP start="0" colour="ffff0000"/>
```

Reduce Boilerplate with CachedValues

- Use `CachedValue<>` to remove accessor/mutator methods
- Easily add undo/redo by supplying an `UndoManager` (3rd argument)
- Still type-safe
- We've lost range checking!

```
struct Clip
{
    Clip (const ValueTree& v)
        : state (v)
    {
        jassert (v.hasType (IDs::CLIP));
        start.referTo (state, IDs::start, nullptr);
        colour.referTo (state, IDs::colour, nullptr);
    }

    ValueTree state;
    CachedValue<double> start;
    CachedValue<Colour> colour;
};
```

```
template<>
struct VariantConverter<Colour>
{
    static Colour fromVar (const var& v)
    {
        return Colour::fromString (v.toString());
    }

    static var toVar (const Colour& c)
    {
        return c.toString();
    }
};
```

```
ValueTree clipState (IDs::CLIP);
Clip c (clipState);
c.start = -1.0;
c.colour = Colours::red;
DBG(c.state.toXmlString());

</> Output:
<CLIP start="-1.0" colour="ffff0000" />
```

Add Verification with Wrapper Classes

```
template<typename Type, typename Constrainer>
struct ConstrainerWrapper
{
    ConstrainerWrapper() = default;

    template<typename OtherType>
    ConstrainerWrapper (const OtherType& other)
    {
        value = Constrainer::constrain (other);
    }

    ConstrainerWrapper& operator= (const ConstrainerWrapper& other) noexcept
    {
        value = Constrainer::constrain (other.value);
        return *this;
    }

    bool operator== (const ConstrainerWrapper& other) const noexcept    { return value == other.value; }
    bool operator!= (const ConstrainerWrapper& other) const noexcept    { return value != other.value; }

    operator var() const noexcept                                       { return Constrainer::constrain (value); }
    operator Type() const noexcept                                      { return Constrainer::constrain (value); }

    Type value = Type();
};
```

Add Verification with Wrapper Classes (2)

```
struct StartTimeConstrainer
{
    static double constrain (const double& v)
    {
        return Range<double> (0.0, 42.0).clipValue (v);
    }
};
```

```
struct Clip : public ReferenceCountedObject
{
    Clip (const ValueTree& v) : state (v)
    {
        jassert (v.hasType (IDs::CLIP));
        start.referTo (state, IDs::start, nullptr);
        colour.referTo (state, IDs::colour, nullptr);
    }

    ValueTree state;
    CachedValue<ConstrainerWrapper<double, StartTimeConstrainer>> start;
    CachedValue<Colour> colour;
};
```

```
ValueTree clipState (IDs::CLIP);
Clip c (clipState);
```

```
c.start = 0.0;
DBG("start: " << c.start.get());
```

```
c.start = 10.0;
DBG("start: " << c.start.get());
```

```
c.start = 43.0;
DBG("start: " << c.start.get());
```

```
c.start = -1.0;
DBG("start: " << c.start.get());
```

</> Output:

```
start: 0
start: 10
start: 42
start: 0
```

Add Verification with Wrapper Classes (3)

```
template<typename Type, int StartValue, int EndValue>
struct RangeConstrainer
{
    static Type constrain (const Type& v)
    {
        const Type start = static_cast<Type> (StartValue);
        const Type end = static_cast<Type> (EndValue);

        return Range<Type> (start, end).clipValue (v);
    }
};


CachedValue<ConstrainerWrapper<double, RangeConstrainer<double, 0, 42>>> start;
```

Add Verification with Wrapper Classes (4)

```
struct NumberConstrainer
{
    static String constrain (const String& v)
    {
        return v.retainCharacters ("0123456789");
    }
};
```

```
struct LetterConstrainer
{
    static String constrain (const String& v)
    {
        MemoryOutputStream os;
        auto p = v.getCharPointer();

        do
        {
            if (p.isLetter())
                os << String::charToString (*p);
        }
        while (p.getAndAdvance());

        return os.toString();
    }
};
```

```
ValueTree c (IDs::CLIP);
CachedValue<ConstrainerWrapper<String, LetterConstrainer>> name;
name.referTo (c, IDs::name, nullptr);
name = "He110 W0r1d";
DBG("name: " << name.get());
```

```
</> Output:
name: HeWrd
```

CachedValue<> Wrapper Classes Summary

- It's possible to use templates to specify constraints on data held in `CachedValue<>`
- Alternatively you could create a version of `CachedValue<>` that takes a `std::function<double (double)>` to constrain the contents

Lists of Objects - ValueTreeObjectList<>

- Although it's possible to have all of your classes derive from `ValueTree::Listener`, often you're managing lists of objects
- We have developed `ValueTreeObjectList` to take care of this
- Easier to deal with rather than iterating child trees
  - Easier to iterate
  - More efficient
  - Enforced verification and validation
  - Type-safety
- Avoids lists of mixed types

Lists of Objects - ValueTreeObjectList<> (2)

```
ValueTree track (IDs::TRACK);
track.addChild (ValueTree (IDs::CLIP), -1, nullptr);
track.addChild (ValueTree (IDs::CLIP), -1, nullptr);

// Only if children are all CLIPS!
DBG("Num Clips: " << track.getNumChildren());

// Add another CLIP child
track.addChild (ValueTree (IDs::CLIP), -1, nullptr);
DBG("Num Clips: " << track.getNumChildren());

// Iterate track's child trees
for (auto c : track)
{
    // Have to check type of child
    if (c.hasType (IDs::CLIP))
    {
        Clip clip (c); // Constructing a Clip might be costly!
        clip.setColour (Colours::blue);
    }
}
```

```
ValueTree track (IDs::TRACK);
track.addChild (ValueTree (IDs::CLIP), -1, nullptr);
track.addChild (ValueTree (IDs::CLIP), -1, nullptr);

ClipList clipList (track);
DBG("Num Clips: " << clipList.objects.size());

// Add another CLIP child
track.addChild (ValueTree (IDs::CLIP), -1, nullptr);
DBG("Num Clips: " << clipList.objects.size());

// Call some methods
for (auto c : clipList.objects)
    c->setColour (Colours::blue);
```

Lists of Objects - ValueTreeObjectList<> (3)

- Listens to a single `ValueTree` (parent)
- When a child of a specific type is added there is a virtual method to create an object of that type
- When a child is removed, there is a virtual method to destroy the object referring to that tree
- If the child order is changed, the managed objects are re-ordered to follow
- Object "`ValueTree state`" is used as a unique identifier
- You now have a simple array of objects to utilise that are tied to the tree

```
template<typename ObjectType>
class ValueTreeObjectList   : public juce::ValueTree::Listener
{
public:
    ValueTreeObjectList (const juce::ValueTree& parentTree);
    virtual ~ValueTreeObjectList();

    // Call in the sub-class when being created
    void rebuildObjects();

    // Call in the sub-class when being destroyed
    void freeObjects();

    //==============================================================
    virtual bool isSuitableType (const juce::ValueTree&) const = 0;
    virtual ObjectType* createNewObject (const juce::ValueTree&) = 0;
    virtual void deleteObject (ObjectType*) = 0;

    virtual void newObjectAdded (ObjectType*) = 0;
    virtual void objectRemoved (ObjectType*) = 0;
    virtual void objectOrderChanged() = 0;

    juce::Array<ObjectType*> objects;
    //...
```

Lists of Objects - ValueTreeObjectList<> (4)

```
struct ClipList : public drow::ValueTreeObjectList<Clip>
{
    ClipList (const ValueTree& v)
        : drow::ValueTreeObjectList<Clip> (v)
    {
        rebuildObjects();
    }

    ~ClipList()
    {
        freeObjects();
    }

    bool isSuitableType (const ValueTree& v) const override
    {
        return v.hasType (IDs::CLIP);
    }

    Clip* createNewObject (const juce::ValueTree& v) override
    {
        return new Clip (v);
    }

    void deleteObject (Clip* c) override
    {
        delete c;
    }

    void newObjectAdded (Clip*) override    {}
    void objectRemoved (Clip*) override     {}
    void objectOrderChanged() override      {}
};
```

```
ValueTree track (IDs::TRACK);
track.addChild (ValueTree (IDs::CLIP), -1, nullptr);
track.addChild (ValueTree (IDs::CLIP), -1, nullptr);

ClipList clipList (track);
DBG("Num Clips: " << clipList.objects.size());

// Add another CLIP child
track.addChild (ValueTree (IDs::CLIP), -1, nullptr);
DBG("Num Clips: " << clipList.objects.size());
```

```
// Call some methods
for (auto c : clipList.objects)
    c->setColour (Colours::blue);

DBG(track.toXmlString());
```

</> Output:

```
Num Clips: 2
Num Clips: 3
<?xml version="1.0" encoding="UTF-8"?>

<TRACK>
  <CLIP colour="ff0000ff"/>
  <CLIP colour="ff0000ff"/>
  <CLIP colour="ff0000ff"/>
</TRACK>
```

LIVE_DEMO - 2_MultipleViews (Simple Tracks and Clips)

- ~200 lines new app code

Thread Safety

- `ValueTreeObjectList<>` is not thread safe
- If you need to access objects from other threads, use reference counting
- Use `ReferenceCountedObject` as a base class for your objects
- Inc the refcount in `createNewObject`, dec it in `deleteObject`
- Ensure no potentially dangling objects are referenced internally
- Keep a `ReferenceCountedArray` of the objects (with `CriticalSection` locking) and return copies of it
- Your method calls still need to be thread safe!

```
struct ClipList : private drow::ValueTreeObjectList<Clip>
{
    ClipList (const ValueTree& v)
        : drow::ValueTreeObjectList<Clip> (v)
    {
        rebuildObjects();
    }

    ~ClipList()                                         { freeObjects(); }

    /** Returns the clips in a thread-safe way. */
    ReferenceCountedArray<Clip> getClips() const    { return clips; }

private:
    bool isSuitableType (const ValueTree& v) const override
    {
        return v.hasType (IDs::CLIP);
    }

    Clip* createNewObject (const juce::ValueTree& v) override
    {
        auto c = new Clip (v);
        clips.add (c);
        return c;
    }

    void deleteObject (Clip* c) override
    {
        clips.removeObject (c);
    }

    void newObjectAdded (Clip*) override    { /** Sort clips */ }
    void objectRemoved (Clip*) override     { /** Sort clips */ }
    void objectOrderChanged() override      { /** Sort clips */ }

    ReferenceCountedArray<Clip, CriticalSection> clips;
};
```

Thread Safety (2)

- For full thread safety provide access to the lock
- Avoids creating a copy of the array
- Or implement a visitor method to enforce the lock
- STILL NOT REAL-TIME SAFE due to priority inversion on the lock!
- For real-time access take a copy of `ReferenceCountedArray` when you construct your real-time object on the message thread

```
struct ClipList : private drow::ValueTreeObjectList<Clip>
{
    //...

    /** Returns the clips in a thread-safe way. */
    const ReferenceCountedArray<Clip, CriticalSection>& getClips() const
    {
        return clips;
    }

private:
    //...

    ReferenceCountedArray<Clip, CriticalSection> clips;
};
```

```
struct ClipList : private drow::ValueTreeObjectList<Clip>
{
    //...

    template<typename Visitor>
    void visitClips (Visitor&& v)
    {
        const ScopedLock sl (clips.getLock());

        for (auto c : clips)
            v (c);
    }

private:
    //...

    ReferenceCountedArray<Clip, CriticalSection> clips;
};
```

CachedValue<> Thread Safety

- `CachedValue<>` is not thread safe
- However, using the wrapper trick from before we can create an atomic wrapper around primitive types
- This is only thread-safe for reading on different threads, writing can still only be done on the message thread as it needs to set the `ValueTree` property
- This combined with the previous `ValueTreeObjectList` techniques enable you to have thread safe reads from `ValueTree` managed objects

```
template<typename Type>
struct AtomicWrapper
{
    AtomicWrapper() = default;

    template<typename OtherType>
    AtomicWrapper (const OtherType& other)
    {
        value.store (other);
    }

    AtomicWrapper (const AtomicWrapper& other)
    {
        value.store (other.value);
    }

    AtomicWrapper& operator= (const AtomicWrapper& other) noexcept
    {
        value.store (other.value);
        return *this;
    }

    bool operator== (const AtomicWrapper& other) const noexcept
    {
        return value.load() == other.value.load();
    }

    bool operator!= (const AtomicWrapper& other) const noexcept
    {
        return value.load() != other.value.load();
    }

    operator var() const noexcept       { return value.load(); }
    operator Type() const noexcept      { return value.load(); }

    std::atomic<Type> value { Type() };
};
```

Summary

- `ValueTree` is like XML with a listener interface
- Automatically handles serialisation and undo/redo
- Complex data can be stored by serialising to/from `String` or `MemoryBlocks`
- Ensure type safety by creating wrappers around `ValueTrees`
- Avoid accessors/mutators by using `CachedValue<>`
- Create lists automatically managed by `ValueTree` state using `ValueTreeObjectList<>`
- Enforce thread-safety & verification by using `CachedValue<WrapperType<>>`

Future Thoughts

- `ValueTrees` can be kept in sync using a `ValueTreeSynchroniser`
  - Can keep apps on different devices in sync e.g. desktop/tablet
  - Leads to interesting ideas about remote clients such as cloud...
- Performance improvements
  - Copy-on-write memory blocks?

Questions?

Presentation/notes available on github:
https://github.com/drowaudio/presentations