

Utilisation de JUCE ValueTrees et du C++ moderne pour créer des applications à grande échelle

David Rowland Tracktion

(traduit en Français, j'ai volontairement gardé certains termes anglophone comme callback ce qui me semblait préférable à la compréhension)

Note:

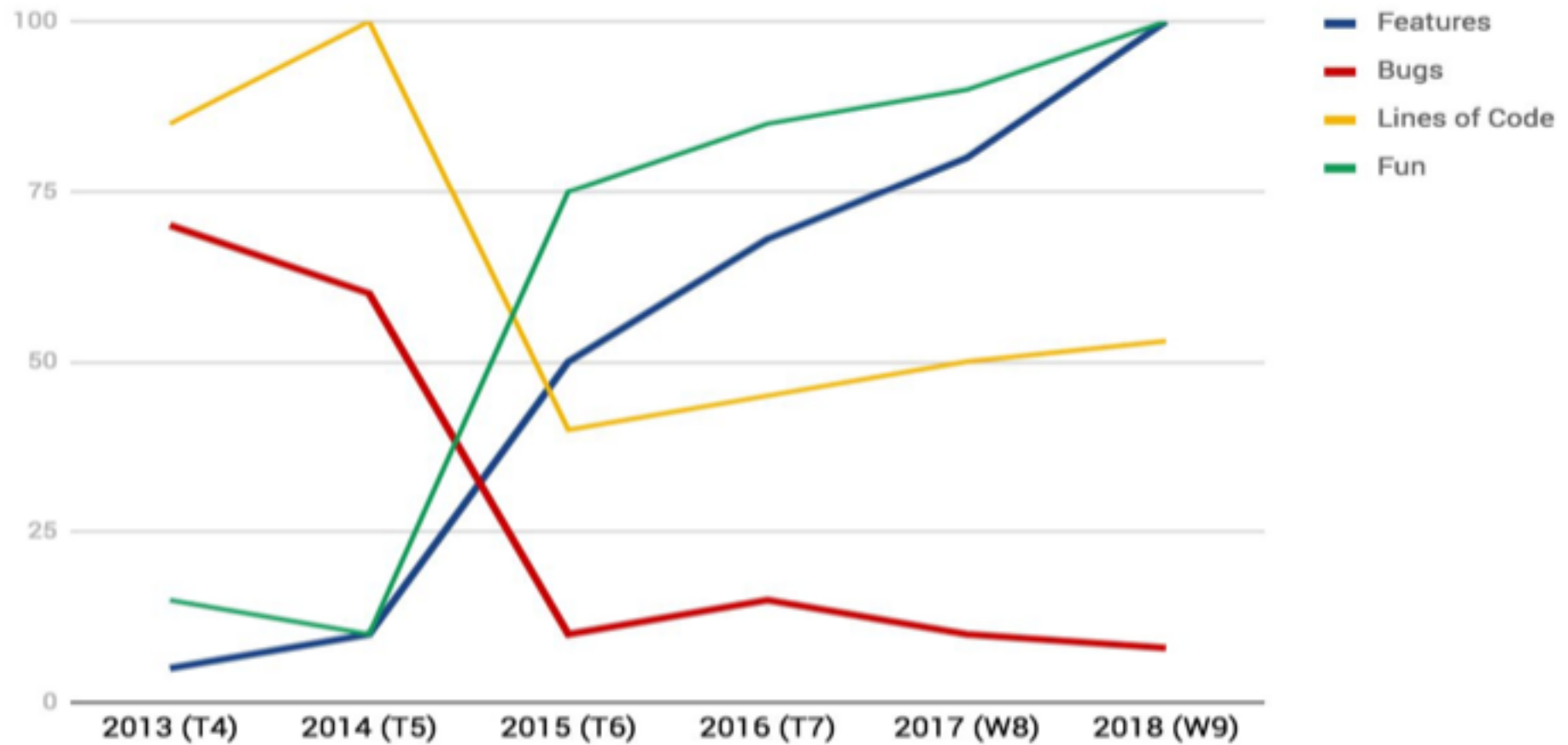
- Les diapositives sont numérotées
- Diapositives/exemples seront sur GitHub
- Le style du code a été modifié pour la présentation des diapositives

1.Introduction

Pourquoi ValueTree ?

- Il y a 4 ans, nous avons restructuré Tracktion pour utiliser ValueTrees.
- Refondu 1/2 millions de lignes de code de base
- Maintenant ~1/4 million de lignes

Evolution de tracktion au fil du temps:



Intro:

- Vue d'ensemble des classes liées à ValueTree
 - var, Identifier, ValueTree, ValueTree, Value
- Regard approfondi sur ValueTree
 - Rappels, lifetime, reference counting, sécurité des threads
 - Stockage de données personnalisé, sérialisation, annulation/répétition (undo/redo)
- Utilisation de ValueTrees comme modèles de données d'application (**MVC**)
- Construire des listes d'objets sécurisées par type

2 Explication des classes var, ValueTree et Value

Juce::Var :

Type var

- Analogue à une var Javascript
- Peut être utilisé pour stocker :
 - Types primitifs (int, int64, bool, double)
 - Juce (String, MemoryBlock)
 - Juce::Tableaux de vars Array<var>
 - juce::ReferenceCountedObjects
 - Callable juce::DynamicObjects
- Peut être sérialisé avec JSON
- Conversion entre types à l'aide des opérateurs compétents

```
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 est comme un enum

- En fait, il s'agit d'un ensemble de chaînes de caractères (String) globalement mise en commun.
- Les comparaisons entre les identificateurs sont des comparaisons de pointeurs donc extrêmement rapides
- La création d'un identificateur peut être lente ($O(\log n)$ où n est le nombre de String)
- La meilleure façon de les créer est statiquement donc ils sont créés au démarrage de l'application

Suggestion: Utilisez un ensemble d'identificateurs global et une macro pour la créer (utiliser le même nom que l'ID pour la lisibilité et éviter les fautes d'orthographe)

```
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 est analogue à XML
- Sa structure est la même (sans texte)
- Imaginez si vous pouviez avoir un XmlElement avec un Listener quand les attributs sont changés et quand les enfants (child sous éléments) sont ajoutés/supprimés/déplacés. Et bien c'est ça juce::ValueTree !
- Terminologie :
 - 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 - Compteur de reference (reference counting)


- Les ValueTrees sont en fait des objets partagés (SharedObject) encapsulé dans un objet doté d'un compteur de référence.
- Copier un ValueTree incrémente simplement le compteur de référence du SharedObject.
- Cela signifie qu'il est peu coûteux de stocker et de copier des ValueTrees puisqu'ils pointent vers des données partagées.
- Vous pouvez créer une copie unique en utilisant 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:
```

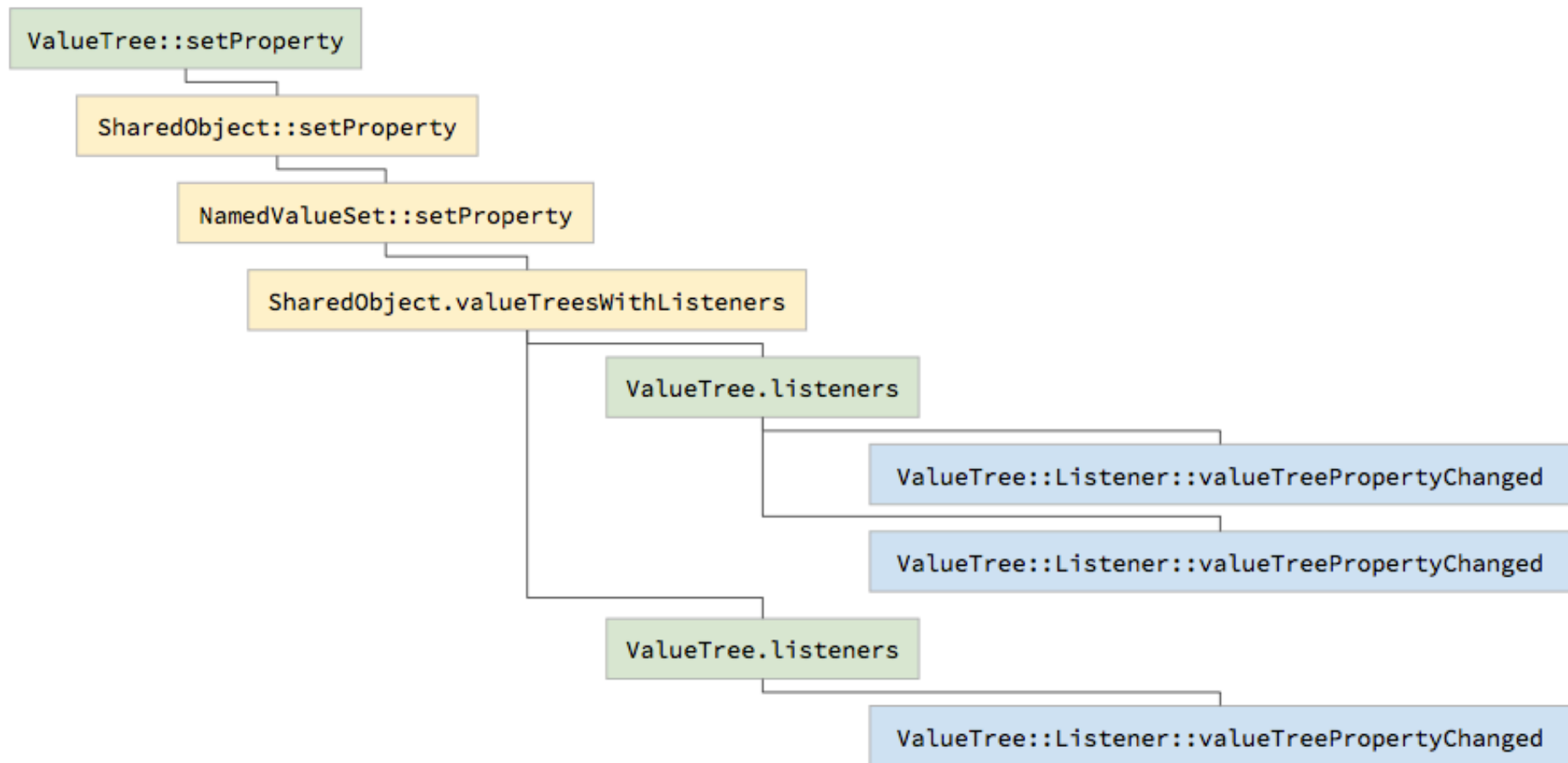
```
1. Ref count v1: 1  
2. Ref count v1: 2  
3. Ref count v2: 2  
4. Ref count v3: 1
```

Juce::ValueTree - Callbacks (1)


Les ValueTrees envoient des Callbacks quand :

- Les propriétés changent
- Les enfants sont ajoutés/supprimés/déplacés
- Le parent change
- Celles-ci sont essentiellement envoyées depuis des modifications apportées à l'objetPartagé (sharedObject) interne.
- De plus, il y a un Callback pour être averti lorsqu'un ValueTree est réassigné, c'est à dire quand le SharedObject est modifié.
-  Les rappels sont synchrones


juce::ValueTree - Callbacks (2)



Juce::ValueTree - Callbacks (3)

- Lorsque vous vous inscrivez en tant que Listener, le ValueTree contient un pointeur vers le Listener, et non le SharedObject
- Cela signifie qu'il est généralement préférable de prendre une copie du ValueTree et de s'enregistrer auprès de ce dernier.
-  Les Callbacks s'ordonnent vers le haut dans la hiérarchie de l'arbre pour que les auditeurs des nœuds parents reçoivent les Callbacks de changement de propriété pour les arbres profondément imbriqués.
- Assurez-vous de concevoir votre arborescence en conséquence et de vérifier les types/propriétés dans chacun de vos Callbacks.

Juce::ValueTree - Callbacks (4)

-  Ne vous fiez pas à l'ordre des rappels, c.-à-d. ne présumez pas qu'un rappel aura eu lieu avant un autre pour la même propriété.
- Utilisez un AsyncUpdater pour vous assurer que tous les rappels ont eu lieu et que vos objets concrets ont le bon état avant de les utiliser.


Juce::ValueTree - Undo/redo

- Parce que tous les changements apportés à un ValueTree sont soit des changements de propriété, soit des ajouts/suppressions/déplacement d'enfants, ils peuvent très facilement être annulés/refaits simplement en ajoutant un juce::UndoManager à ces actions
- Si toutes vos 'vues' répondent simplement à l'état de l'arbre et sont mises à jour quand il change, tout restera toujours synchronisé.

```
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 a des similitudes avec juce::ValueTree mais encapsule un seul juce::var
- En interne, il dispose d'une ValueSource qui sert par défaut de compteur de référence pour le var
- Vous pouvez écouter les changements apportés au var
-  Value Les callbacks sont asynchrones*.
- Utile pour le lier à l'interface utilisateur

```
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)

- Vous pouvez également créer une ValueSource personnalisée que vous pouvez utiliser pour obtenir des messages de changement synchrones.
- Ou encapsuler des données personnalisées sous forme de var
- Vous pouvez créer des Values à partir de propriétés d'un ValueTree
- Ceci crée une ValueSource personnalisée qui est un ValueTree::Listener

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

```

Stocker de données personnalisées

- Vous pouvez stocker des données personnalisées en les convertissant vers ou à partir de Strings ou MemoryBlocks.
- Puisqu'il peut être lent de faire ces conversions, vous pouvez utiliser une CachedValue
- En spécialisant les conversions de var, vous ne faites la conversion de chaîne que lorsque les données changent réellement.
- Certaines données sont externes à votre modèle et nécessiteront un rinçage (flushing) de l'arbre avant la sérialisation, par exemple l'état de l'AudioProcessor.

```
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);
    }
};
```

Utiliser `CachedValue` <> encapsuler les propriétés comme objects (1)

- Permet les méthodes get/set de type "property".
- Des primitives simples, faciles :
 `CachedValue<float> start ;`
 `start.referTo (v, IDs::start, um) ;`
- Techniquement, pas sans danger pour les Threads....

Utiliser CachedValue <> encapsuler les propriétés comme objects (2)

- Les objets plus complexes peuvent être stockés sous forme de chaînes ou de MemoryBlocks encapsuler dans un var.

- Spécialiser VarienConverter<> pour le faire de manière transparente
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 ImageFormat::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);
    }
};
```

Sérialisation (sauvegarde)

- juce::ValueTree peut être facilement sérialisé en XML, un format binaire ou un format binaire compressé.
 - XML est en texte clair et peut donc être débogué
 - Légèrement plus lent que le binaire cependant car il devra être converti en un juce::XmlElement d'abord
 - Le type des objets sont perdues (seront rechargées comme String)
- Le binaire sera le plus rapide pour lire/écrire et les types de var sont correctement sauvegardés/chargés car les opérations juce::var stream sont utilisées pour écrire les marqueurs de type
- Le binaire compressé est identique au format binaire mais peut prendre moins de place.
 - Lecture/écriture plus lente

```
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 sûreté (1)

- `juce::var` n'est pas sans danger pour les threads
- `juce::La` valeur n'est pas sans danger pour les threads
- `juce::ValueTree` n'est pas sans danger pour les threads
- `juce::CachedValue<>` n'est pas sans danger pour les threads
- (Data race on a POD type which could be argued is a benign data-race)
- Ce qui est encore mauvais !
- Assurez-vous que toutes les interactions du `ValueTree` arrivent dans le thread ! Sinon, vos Callback se feront à partir du thread émetteur et feront des ravages.

Thread sûreté (2)

- Utilisez l'envoi de messages, AsyncUpdater, etc. pour la renforcer.
- Ceci devrait garantir que tous les callbacks arrivent dans le thread de message.
- Faites des copies des données et sécurisez ce thread avec des « atomics » ou des « locks »
- `CachedValue<AtomicWrapper<Type>>>`

ValueTrees - Top 5 Things to Know

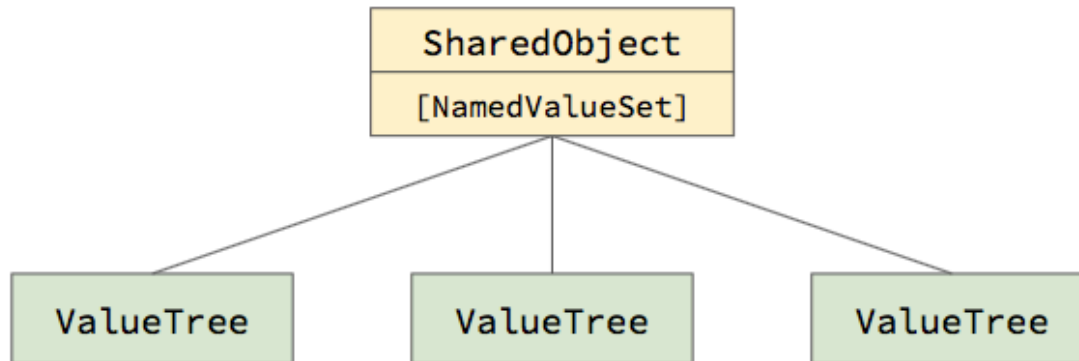
1 ValueTree sont comme des XML

- juce::ValueTree est analogue à XML
- Sa structure est la même (sans texte)
- Imaginez si vous pouviez avoir un XmlElement avec un Listener quand les attributs sont changés et quand les enfants (child sous éléments) sont ajoutés/supprimés/déplacés. Et bien c'est ça juce::ValueTree !
- Terminologie :
 - 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="29.0768939999999999352"
      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 ValueTree sont encapsulé dans un SharedObject

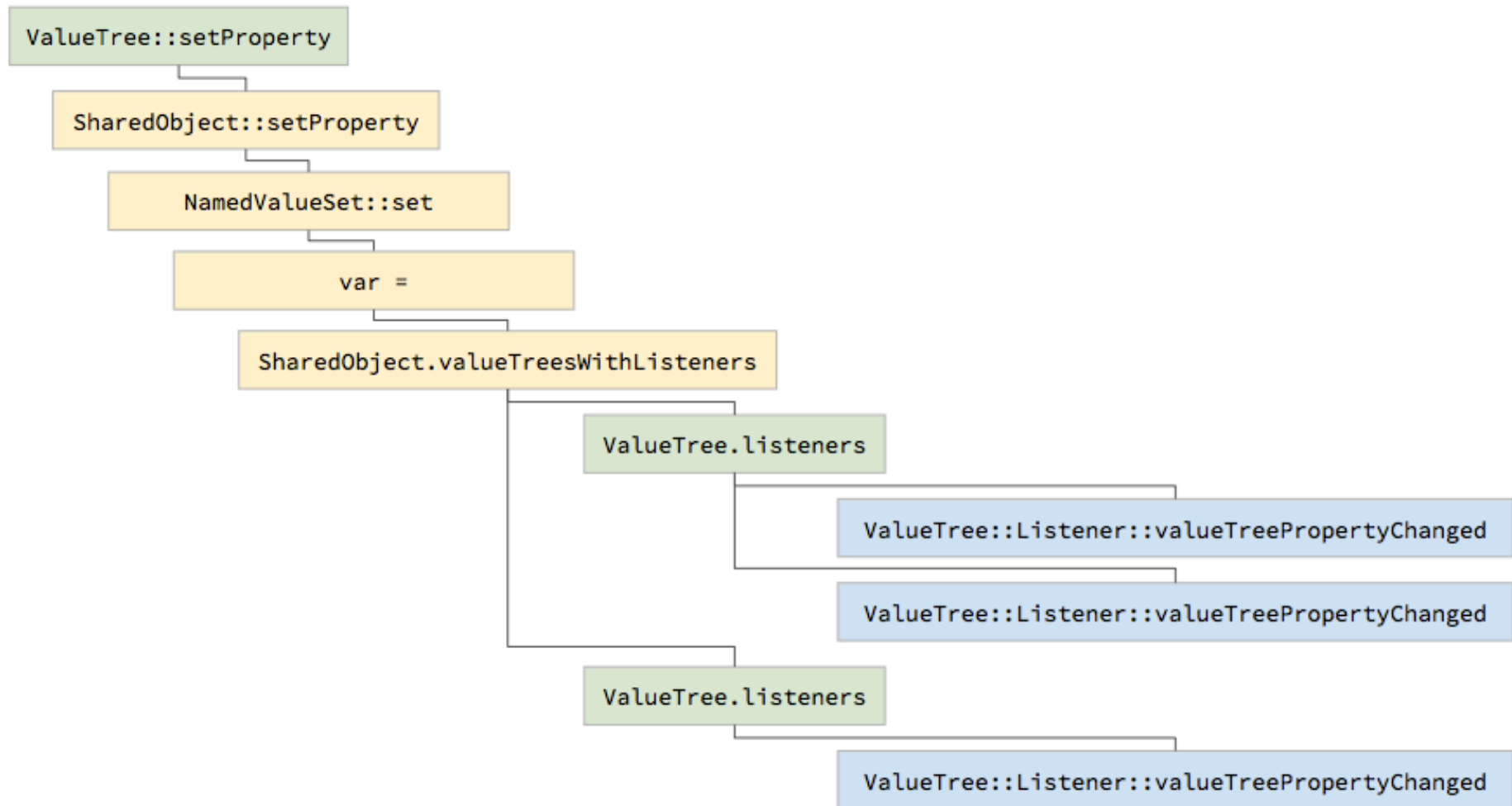
- Les ValueTrees sont des objets légers qui gèrent un reference-counted SharedObject
- Le SharedObject maintient un NamedValueSet des propriétés
- ValueTree fournit une interface pour le SharedObject
- Il est peu coûteux de copier autour de ValueTrees

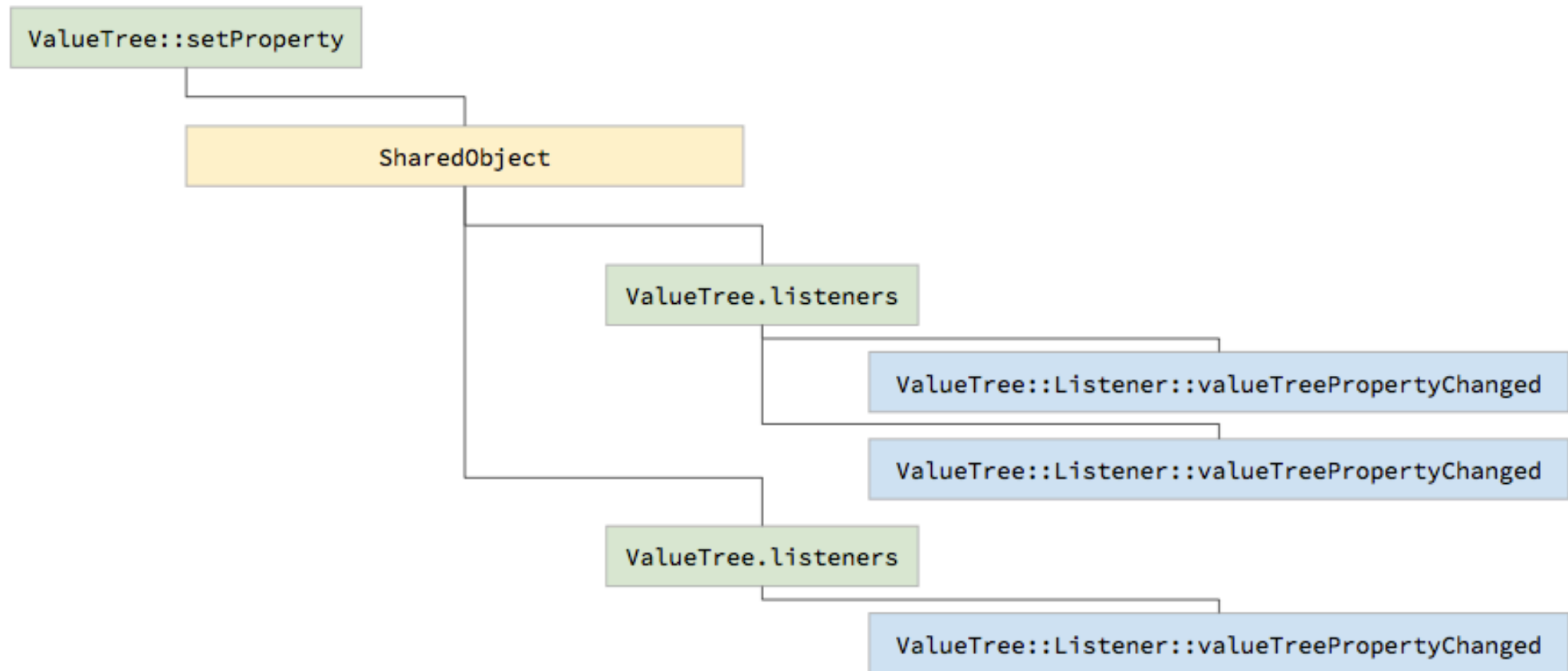


3 Les listeners sont conservés avec le ValueTree

- Lorsque vous définissez un Listener, le ValueTree contient un pointeur vers le Listener, et non vers le SharedObject
- Cela signifie qu'il est généralement préférable de prendre une copie du ValueTree et de s'enregistrer auprès de ce dernier.

```
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 {}
};
```





4 Les callbacks sont synchrone et envoyées à tous les enfants

- Les callbacks sont synchrones, alors ne faites rien qui prenne du temps.
- Utilisez AsyncUpdater pour combiner les actions des changements de propriétés
- Les callback s'ordonnent vers le haut dans la hiérarchie de l'arborescence pour que les auditeurs des nœuds parents reçoivent des rappels de changement de propriété pour les arbres profondément imbriqués.
- Assurez-vous de concevoir votre arborescence en conséquence et de vérifier les types/propriétés dans chacun de vos 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)

    }
    //... };
};
```


5 Sérialiser les données non primitives en chaîne de caractères

- Combiner plusieurs propriétés (properties) liées en un seul « property »
- Sérialiser de/vers « String » une chaîne de caractères
- Pensez à l'efficacité ici, la plupart du temps les propriétés ne changeront pas très souvent. Pour les propriétés qui le font, vous aurez peut-être besoin d'un paradigme légèrement différent.

```
• TimestretchOptions options;  
• options.synchroniseTimePitch = 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:  
01111128
```

```
struct TimestretchOptions  
{  
    TimestretchOptions() = default;  
    TimestretchOptions (const String& s)  
    {  
        auto tokens = StringArray::fromTokens (s, "|", "");  
        stereoMS      = tokens[0].getIntValue() != 0;  
        synchroniseTimePitch = tokens[1].getIntValue() != 0;  
        preserveFormants  
        envelopeOrder  
String toString() const  
    {  
        = tokens[2].getIntValue() != 0;  
        = tokens[3].getIntValue();  
    }  
  
    StringArray s;  
    s.add (stereoMS ? "1" : "0");  
    s.add (synchroniseTimePitch ? "1" : "0");  
    s.add (preserveFormants ? "1" : "0");  
    s.add (String (envelopeOrder));  
    return s.joinIntoString ("|");  
}  
  
bool stereoMS = false, synchroniseTimePitch = false,  
    preserveFormants = false;  
int envelopeOrder = 64;  
};
```


3. Utilisation de ValueTrees comme modèle d'application (MVC)

Trees comme class

- Les ValueTrees s'adaptent naturellement à la nature hiérarchique de nombreuses applications.
- Tree types sont analogues aux noms des classes.
- Les propriétés sont analogues à celles d'une variable

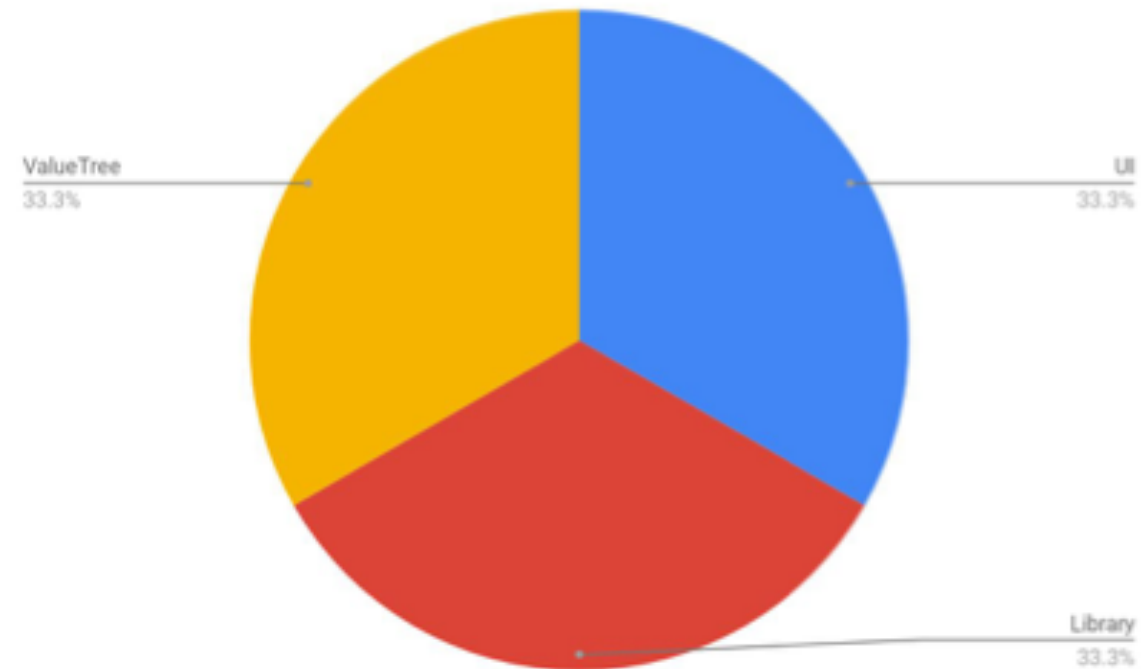
```
<EDIT appVersion="Waveform 8.1.8" mediaId="e31b7/1d17644"
  creationTime="1402484148538" fps="24" timecodeFormat="beats">
  <VIEWSTATE viewleft="-1" viewright="194.9998083000000122"/>
  <TRACK mediaId="e2b7f/23eff3ab" name="melody"
height="34.214285714285715301"
  colour="ffff4d4c">
    <CLIP mediaId="e31b7/d45f454" name="xxxx" type="midi"
source="e2b7f/23eff3c4"
  start="59.0768640000000000487"
length="29.0768939999999999352"
  offset="0" colour="ffffab00"/>
    <CLIP mediaId="e31b7/30291232" name="melodyoutro"
type="midi" source="e2b7f/23eff3c4"
  start="147.6921600000000000122"
length="29.5384320000000000244"
  offset="0" colour="ffffab00"/>
  </TRACK>
</EDIT>
```

Trees et MVC (Modèle de donnée, Vue, Contrôleur)

- En utilisant le mécanisme de callback de ValueTree, vous pouvez construire des objets
- Dans un paradigme de type MVC, le ValueTree prend en charge les aspects du modèle de données et du contrôleur.
- Vous pouvez ensuite écrire plusieurs vues pour représenter ce modèle de données en laissant les callbacks vous avertir lorsque les choses ont changé.
- En apportant des modifications à l'arborescence, les différentes vues se mettent à jour pour refléter le nouvel état de l'arborescence.
- Vous pouvez avoir plusieurs vues si nécessaire.

Live démo 1 ValueTreeDemo

- ~750 lignes de code
 - Arborescence des données
 - Panneau (vue) des propriétés sélectionnées
 - Editeur des couleurs
 - Glisser-déposer (drag and drop)
 - Annuler/refaire (undo / redo)
 - Sérialisation
- ~250 UI code
 - Dessin
 - Panneau de propriétés
- ~250 méthodes d'assistance (helpers) et de bibliothèque réutilisables
 - Enregistrer ValueTree de/vers un fichier
 - juce::Component assistant
- ~250 d'interaction avec ValueTree
 - Création
 - Validation
 - Obtenir (get) /définir (set) les propriétés



Undo / redo

- Lors de la mutation d'un ValueTree, il suffit de fournir un UndoManager pour rendre cette action annulable
- `v.setProperty (IDs::start, 42.0, &undoManager) ;`
- `Clip ValueTree (IDs::CLIP) ;`
`v.addChild (clip, -1, &undoManager) ;`

4. Utilisation avec des applications de grande taille

Objets typé

- Interagir directement avec un ValueTree est rapide, mais il y a des pièges :

- Pas de contrôle de type
- Pas de vérification des limites
- Nécessite des connaître les propriétés des données et de la structure de l'arborescente

- Nous aimons :

- Abstractions
- Sécurité du type
- Efficacité

- Vous pouvez construire des wrappers (encapsuler) autour de ValueTrees pour renforcer tout cela

- `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"/>`

`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;
```

Reduire la taille du code avec les CachedValues

- Utilisez CachedValue<> pour supprimer les méthodes accessor/mutator
- Ajouter facilement undo/redo en fournissant un UndoManager (3ème argument)
- Toujours sûr pour le type
- Nous avons perdu le contrôle de portée !

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

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

```
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;
};
```

```
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" />
```

Ajouter une vérification avec une classe d'encapsulation (1)

```
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();
};
```


Ajouter une vérification avec une classe d'encapsulation (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);
    } </> Output:
```

```
    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());
start: 0
```

```
start: 10
```

```
start: 42
```

```
start: 0
```

Ajouter une vérification avec une classe d'encapsulation (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;
```

Ajouter une vérification avec une classe d'encapsulation (4)

```
struct NumberConstrainer struct LetterConstrainer
{
    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

Liste d'objets - ValueTreeObjectList<> (1)

- Bien qu'il soit possible de faire dériver toutes vos classes de ValueTree::Listener, vous gérez souvent des listes d'objets.
- Nous avons développé ValueTreeObjectList pour répondre à ce besoin.
- Il est plus facile de s'occuper des arbres enfants que de les itérer.
 - Plus facile à itérer
 - Plus efficace
 - Vérification et validation forcées
 - Sécurité du type
- Évite les listes de types mixtes

Liste d'objets - 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)
{
    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);
    // 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);
```

Liste d'objets - ValueTreeObjectList<> (3)

- Ecoute un seul ValueTree (parent)
- Quand un enfant d'un type spécifique est ajouté
il existe une méthode virtuelle pour créer un objet de ce type
- Lorsqu'un enfant est enlevé, il y a une méthode virtuelle pour détruire l'objet se référant à cet arbre
- Si l'ordre des enfants est modifié,
Les objets gérés sont réordonnés et suivit
- L'objet "ValueTree state" est utilisé en tant qu'objet
identifiant unique
- Vous disposez maintenant d'un simple tableau (array) d'objets
à utiliser qui sont liés à l'arbre

```
template<typename ObjectType>
class ValueTreeObjectList
{
public:
    : public juce::ValueTree::Listener

    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;
    //...
```

Liste d'objets - 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 Vues multiple (simple pistes et clips)

~200 lignes de nouveau code d'application

Thread sûreté (1)

- ValueTreeObjectList<> n'est pas sûr pour le thread
- Si vous avez besoin d'accéder à des objets à partir d'autres threads, utilisez le comptage de référence
- Utilisez ReferenceCountedObject comme classe de base pour vos objets
- Incrémentez refcount dans createNewObject, décrémentez le dans deleteObject
- S'assurer qu'aucun objet potentiellement suspendu n'est référencé en interne.
- Conservez un tableau de référence des objets (avec verrouillage CriticalSection) et renvoyez-en des copies.
- Vos appels de méthode doivent toujours être sécurisés !

```
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
    void objectRemoved (Clip*) override
    void objectOrderChanged() override
    { /** Sort clips */ }
    { /** Sort clips */ }
    { /** Sort clips */ }
    ReferenceCountedArray<Clip,
        CriticalSection>
        clips;

};
```

Thread sûreté (2)

- Pour une sécurité totale du thread, donnez accès au lock.
- Évitez de créer une copie du tableau
- Ou mettre en place une méthode visiteur pour faire respecter le lock
- TOUJOURS PAS SÛR EN TEMPS RÉEL en raison de la priorité du lock !
- Pour un accès en temps réel, prenez une copie de
ReferenceCountedArray lorsque vous construisez votre objet en temps réel sur le thread de message

```
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 sûreté

- CachedValue<> n'est pas sûr pour les threads
- Cependant, en utilisant l'astuce d'encapsulation avant de pouvoir créer un atomic wrapper autour des types primitifs (int, int64, float, double ...)
- Ceci n'est sans danger que pour la lecture de différents threads, l'écriture ne peut toujours se faire que sur le thread message car il faut définir ValueTree property
- Combiné avec les techniques précédentes de ValueTreeObjectList, cela vous permet d'avoir des lectures sécurisées à partir d'objets gérés par ValueTree.

```
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() };
};
```

Résumé

- ValueTree est comme XML avec une interface d'écoute (Listener)
- Prise en charge automatique de la sérialisation et de l'annulation/répétition (undo/redo).
- Des données complexes peuvent être stockées en sérialisant des données de chaîne ou de chaîne de caractères.

MemoryBlocks

- Assurer la sécurité des types en encapsulant autour de ValueTrees

- Évitez les accesseurs/mutateurs en utilisant CachedValue<>.

- Créer des listes gérées automatiquement par ValueTree state à l'aide de la commande

ValueTreeObjectList <>

- Renforcer la sécurité et la vérification des threads en utilisant CachedValue<WrapperType<>>.

Réflexions pour le futur

- Les ValueTrees pourraient être synchronisés à l'aide d'un ValueTreeSynchroniser
 - Possibilité de synchroniser les applications sur différents périphériques, par ex. bureau/tablette
 - Conduit à des idées intéressantes sur les clients distants tels que le cloud....
- Amélioration des performances
 - Blocs de mémoire Copy-on-write ?

Questions?

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

