# Maspack Reference Manual

---

**John Lloyd**
July 8, 2014

# Contents

# 1 Introduction

Maspack (modeling and simulation package) is a set of Java packages to support physical modeling and simulation.

# 2 Properties

The maspack property package provides a uniform means by which classes can export specific attributes and information about them to application software. The main purpose of properties is to

1. Provide generic code for accessing and modifying attributes.
2. Remove the need for "boiler-plate" code to read or write attributes from persistent storage, or modify them by other means such as a GUI panel.

The property software uses Java reflection to obtain information about a property's value and its associated class, in a manner similar to that used by the properties of Java Beans.

## 2.1 Accessing Properties

Any class can export properties by implementing the interface HasProperties:

```
interface HasProperties
{
   // get a handle for a specific named property
   Property getProperty (String name);

   // get a list of all properties associated with this class
   PropertyInfoList getAllPropertyInfo ();
}
```

Each property is associated with a name, which must be a valid Java identifier. The getProperty() method returns a Property handle to the named property, which can in turn be used to access that property's values or obtain other information about it. getAllPropertyInfo() returns a PropertyInfoList providing information about all the properties associated with the class.

A Property handle supplies the following methods:

```
interface Property
{
   Object get();
   void set (Object value);
   Range getRange ();
   HasProperties getHost();
   PropertyInfo getInfo();
}
```

`get()`

Returns the property's value. As a rule, returned returned values should be treated as read-only.

`set()`

Sets the property's value (unless it is read-only, see Section 2.2).

`getRange()`

Returns a `Range` object for a property (see Section 2.1.1), which is used to determine which values are admissible to `set`. If all values are admissible, `getRange()` can return `null`.

`getHost()`

Returns the object to which this property handle belongs.

`getInfo()`

Returns static information about this property (see Section 2.2).

A simple example of property usage is given below. Assume that we have a class called `DeformableModel` which contains a property called `stiffness`, and we want to set the stiffness to 1000. This could be done as follows:

```
DeformableModel defo = new DeformableModel();
Property stiff = defo.getProperty("stiffness");

stiff.set (1000.0); // (uses Java 1.5 autoboxing to turn
                    // 1000.0 into Double(1000.0)
```

Of course, `DeformableModel` will likely have a method called `setStiffness` that can be used to set the stiffness directly, without having to got through the `Property` interface. However, the purpose of properties is not to facilitate attribute access within specially constructed code; it is to facilitate attribute access within *generic* code that is hidden from the user. For instance, suppose I want to query a property value from a GUI. The GUI must obtain the name of the desired property from the user (e.g., through a menu or a text box), and then given only that name, it must go and obtain the necessary information from the object exporting that property. A `Property` allows this to be done in a manner independent of the nature of the property itself.

### 2.1.1 Why Property Handles?

In theory, one could embed the methods of `Property` directly within the `HasProperties` interface, using methods with signatures like

```
Object getPropertyValue (String name);

void setPropertyValue (String name, Object value);
```

The main reason for not doing this is performance: a property handle can access the attribute quickly, without having to resolve the property's name each time.

Each property handle contains a back-pointer to the object containing, or *hosting*, the property, which can be obtained with the `getHost()` method.

## 2.2 Property Ranges

A Range object supplies information about what values a particular `Property` can be set to. It contains the following methods:

```
interface Range
{
   boolean isValid (Object obj, StringHolder errMsg);
   Object projectToRange (Object obj);
   void intersect (Range range);
   boolean isEmpty();
}
```

`isValid()`

Returns `true` if `obj` is a valid argument to the property's `set` method. The optional argument `errMsg`, if not `null`, is used to return an error message in case the object is not valid.

`makeValid()`

Trys to turn `obj` into a valid argument for `set()`. If `obj` is a valid argument, then it is returned directly. Otherwise, the method tries to return an object close to `obj` that is in the valid range. If this is not possible, the method returns `Range.IllegalValue`.

`intersect()`

Intersects the current range with another range and placed the result in this range. The resulting range should admit values that were admissible by both previous ranges.

`isEmpty()`

Returns `true` if this range has no admissible values. This is most likely to occur as the result of an intersection operation.

Possible usage of a range object is shown below:

```
   Property prop = hostHost.get ("radius");
   Range range = prop.getRange();
   StringHolder errMsg = new StringHolder();
   double r;

   ...

   if (!range.isValid (r, errMsg)) {
      System.err.println ("Radius r invalid, reason: " {\tt  errMsg.value);
   }
   else {
      prop.set (r);
   }
```

Two common examples of `Range` objects are DoubleInterval and IntegerInterval, which implement intervals of double and integer values, respectively. Ranges are also `Clonable`, which means that they can be duplicated by calling `range.clone()`.

## 2.3 Obtaining Property Information

Additional information about a property is available through the PropertyInfo interface, which can be obtained using the `getInfo()` method of the property handle. Information supplied by `PropertyInfo` is static with respect to the exporting class and does not change (unlike the property values themselves, which do change). Such information includes the property's name, whether or not it is read-only, and a comment describing what the property does.

Some of the `PropertyInfo` methods include:

```
interface PropertyInfo
{
   // gets the name of this property
   String getName();

   // returns true if this property cannot be set
   boolean isReadonly();

   // returns a string description of the property
   String getDescription();

   // returns an optional format string describing how the
   // property's values should be formatted when printed.
   String getPrintFormat();

   // returns the class associated with this property's value.
   Class getValueClass();

   // returns the class associated with this property's host
   Class getHostClass();

   // returns true if the properties value should be written
   // by the PropertyList write method.
   boolean getAutoWrite();
```

```
   // returns the conditions under which this property
   // should be interactively edited.
   Edit getEditing();

   // Returns information about whether the property's editing widget
   // should be able to expand or contract in order to save GUI space.
   ExpandState getWidgetExpandState();

   // returns the default value for this property
   Object getDefaultValue();

   // returns a default numeric range for this property, if any
   public NumericInterval getDefaultNumericRange();

   // writes a value of this object out to a PrintStream
   void writeValue (
       Object value, PrintWriter pw, NumberFormat fmt);

   // scans a value of this object from a ReaderTokenizer
   Object scanValue (ReaderTokenizer rtok);

   // creates a Property for this property, attached
   // to the specified host object
   Property createHandle (HasProperties host);

   // returns true if a specified value equals this
   // property's default value
   boolean valueEqualsDefault();

   // returns true if the property is inheritable
   boolean isInheritable();

   // returns the property's numeric dimension, or -1 if it
   // is not numeric or does not have a fixed dimension
   int getDimension();

   // indicates that the property value can be shared among
   // several hosts.
   boolean isSharable();
}
```

Property information can also be obtained directly from the exporting class, using `getAllPropertyInfo()`, which returns information for all the properties exported by that class. This information is contained within a PropertyInfoList:

```
interface PropertyInfoList
{
   // number of properties in this list
   int size();

   // returns an iterator over all the property infos
   Iterator<PropertyInfo> iterator();

   // returns info for a specific named property
   PropertyInfo get (String name);

   // returns true if this list has no inheritable properties
   boolean hasNoInheritableProperties();
}
```

For example, suppose we want to print the names of all the properties associated with a given class. This could be done as follows:

```
HasProperties exportingObject;
...
PropertyInfoList infoList =
    exportingObject.getAllPropertyInfo();
for (PropertyInfo info : infoList) {
   System.out.println (info.getName());
}
```

## 2.4 Exporting Properties from a Class

As indicated above, a class can export properties by implementing the interface `HasProperties`, along with the supporting interfaces `Property`, `PropertyInfo`, and `PropertyInfoList`. The class developer can do this in any way desired, but support is provided to make this fairly easy.

The standard approach is to create a static instance of `PropertyList` for the exporting class, and then populate it with `PropertyInfo` structures for the various exported properties. This `PropertyList` (which implements `PropertyInfoList`) can then be used to implement the `getProperty()` and `getAllPropertyInfo()` methods required by `HasProperties`:

```
protected static PropertyList myProps;

... initialize myProps in a static code block ...

// returns an information list for all properties
public PropertyInfoList getAllPropertyInfo () {
   return myProps;
}

// returns a handle for a specific property
public Property getProperty (String name) {
   getAllPropertyInfo().getProperty (name, this);
}
```

Information about specific properties should be added to `PropertyList` within a static code block (second line in the above fragment). This can be done directly using the method

```
void add (PropertyInfo info)
```

but this requires creating and initializing a `PropertyInfo` object. An easier way is to use a different version of the `add` method, which creates the required `PropertyInfo` structure based on information supplied through its arguments. In the example below, we have a class called `ThisHost` which exports three properties called `visible`, `lineWidth`, and `color`:

```
// default values for the properties
protected static int defaultLineWidth = 1;
protected static boolean defaultVisibleP = true;
protected static Color defaultColor =
   new Color (0.5f, 0.5f, 0.5f);

// fields containing the property values
protected int myLineWidth = defaultLineWidth;
protected boolean myVisibleP = defaultVisibleP;
protected Color myColor = defaultColor;

// create a PropertyList ...
protected static PropertyList myProps =
   new PropertyList(ThisHost.class);

// ... and add information for each property:
static {
   myProps.add (
```

```
         "visible isVisible *", "object is visible",
          defaultVisibleP);
      myProps.add (
         "lineWidth", "line width (pixels)",
         defaultLineWidth);
      myProps.add (
         "color", "color ", defaultColor);
   }

   public PropertyInfoList getAllPropertyInfo () {
      return myProps;
   }

   public Property getProperty (String name) {
      getAllPropertyInfo().get (name, this);
   }
```

The values for the three properties are stored in the fields `myLineWidth`, `myVisibleP`, and `myColor`. Default values for these are defined by static fields.

A static instance of a `PropertyList` is created, using a constructor which takes the exporting class as an argument (in Java 1.5, the class object for a class can be referenced as *ClassName*`.class`). Information for each property is then added within a static block, using the convenience method

```
   void add (String nameAndMethods, String description,
             Object defaultValue)
```

The first argument, `nameAndMethods`, is a string which gives the name of the property, optionally followed by whitespace-separated names of the accessor methods for the property's value:

```
   "<propertyName> [<getMethodName>] [<setMethodName>] [<getRangeMethodName>]"
```

These accessor methods should have the signatures

```
   Object getMethod();

   void setMethod (Object value);

   Range getRangeMethod ();
```

If any of the methods are not specified, or are specified by a '`*`' character, then the system with look for accessor methods with the names `getXxx`, `setXxx`, and `getXxxRange`, where `xxx` is the name of the property. If no `getRangeMethod` is defined (and no numeric range is specfied in the `options` argument string, as described below), then the property will be assumed to have no range limitations and its `getRange()` method will return `null`.

The second argument, `description`, gives a textual description of the property, and is used for generating help messages or tool-tip text.

The third argument, `defaultValue`, is a default property value, which is used for automatic initialization and for deciding whether the property's value needs to be written explicitly to persistent storage.

An extended version of the `add` method takes an additional argument `options`:

```
   void add (String nameAndMethods, String description,
             Object defaultValue, String options)
```

The `options` argument is a sequence of option tokens specifing various property attributes, each of which can be queried using an associated `PropertyInfo` method. Token are separated by white space and may appear in any order. Some have have both long and abbreviated forms. They include:

`NW`, `NoAutoWrite`

> Disables this property from being automatically written using the `PropertyList` methods `write` and `writeNonDefaults` (Section 2.5). Causes the `PropertyInfo` method `getAutoWrite()` to return false.

`AW`, `AutoWrite` **(Default setting)**

Enables this property to be automatically written using the `PropertyList` methods `write` and `writeNonDefaults` (Section 2.5). Causes the `PropertyInfo` method `getAutoWrite()` to return true.

`NE`, `NeverEdit`

Disables this property from being interactively edited. Causes the `PropertyInfo` method `getEditing()` to return `Edit.Never`.

`AE`, `AlwaysEdit` **(Default setting)**

Enables this property to be interactively edited. Causes the `PropertyInfo` method `getEditing()` to return `Edit.Always`.

`1E`, `SingleEdit`

Enables this property to be interactively edited for one property host at a time. Causes the `PropertyInfo` method `getEditing()` to return `Edit.Single`.

`XE`, `ExpandedEdit`

Indicates, where appropriate, that the widget for editing this property can be expanded or contracted to conserve GUI space, and that it is initially expanded. Causes the `PropertyInfo` method `getWidgetExpandState()` to return `ExpandState.Expanded`. This is generally relevant only for properties such as CompositeProperties (Section 2.4.2) whose editing widgets have several sub-widgets.

`CE`, `ContractedEdit`

Indicates, where appropriate, that the widget for editing this property can be expanded or contracted to conserve GUI space, and that it is initially contracted. Causes the `PropertyInfo` method `getWidgetExpandState()` to return `ExpandState.Contracted`. This is generally relevant only for properties such as CompositeProperties (Section 2.4.2) whose editing widgets have several sub-widgets.

`D`**X**, `Dimension`**X**

Sets the numeric dimension of this property to X. The dimension can be queried using the `PropertyInfo` method `getDimension()`. For properties which are non-numeric or do not have a fixed dimension, the dimension will be returned as -1. Note the for some numeric properties, the dimension can be determined automatically and there is no need to explicitly specify this attribute.

`SH`, `Sharable`

Indicates that the property value is not copied internally by the host and can therefore be shared among several hosts. This may improve memory efficiency but means that changes to the value itself may be reflected among several hosts. This attribute can be queried by the `PropertyInfo` method `isSharable()`.

`NV`, `NullOK`

Indicates that the property value may be null. By default, this is false, *unless* the default value has been specified as null. Whether or not a property may be set to null is particularly relevant in the case of CompositeProperties (Section 2.4.2), where one may choose between setting individual sub-properties or setting the entrie structure to null altogether.

`%fmt`

A printf-style format string, beginning with `%`, used to format numeric information for this property's value, either in a GUI or when writing to persistent storage. A good general purpose format string to use is often `"%.6g"`, which specifies a free format with six significant characters.

`[l,u]`

A numeric range interval with a lower bound of `l` and an upper bound of `u`. If specified, this defines the value returned by PropertyInfo.getDefaultNumericRange(); otherwise, that method returns `null`. If a `getRangeMethod` is not defined for the property, and the property has a numeric type, then the default numeric range is returned by the property's Property.getRange() method. The default numeric range is also used to determine bounds on slider widgets for manipulating the property's value, in case the upper or lower limits returned by the Property.getRange() method are unbounded. The symbol `inf` can be used in an interval range, so that `[0,inf]` represents the set of non-negative numbers.

The following code fragment shows an example of using the `option` argument:

```
myProps.add (
   "radius", "radius of the sphere (mm)", defaultRadius,
   "%8.3f [0,100] NE");
);
```

The property named `radius` is given a numeric format string of `"%8.3f"`, a numeric range in the interval $[0, 100]$, and set so that it will not be displayed in an automatically created GUI panel.

### 2.4.1 Read-only properties

A property can be *read-only*, which means that it can be read but not set. In particular, the `set()` for a read-only `Property` handle is inoperative.

Read-only properties can be specified using the following `PropertyList` methods:

```
void addReadOnly (String nameAndMethod, String description);

void addReadOnly (String nameAndMethod, String description,
    String options);
```

These are identical to the `add` methods described above, except that the `nameAndMethod` argument includes at most a get accessor, and there is no argument for specifying a default value.

The method `getAutoWrite()` also returns false for read-only properties (since it does not make sense to store them in persistent storage).

### 2.4.2 Inheriting Properties from a superclass

By default, a subclass of a `HasProperties` class inherits all the properties exported by the class exports all the properties exported by it's immediate superclass.

Alternatively, a subclass can create its own properties by creating it's own `PropertyList`, as in the code example of Section 2.3:

```
// create a PropertyList ...
protected static PropertyList myProps =
   new PropertyList(ThisHost.class);

public PropertyInfoList getAllPropertyInfo () {
   return myProps;
}
```

and none of the properties from the superclass will be exported. Note that it is necessary to redefine `getAllPropertyInfo()` so that the instance of `myProps` specific to `ThisHost` will be returned.

If one wishes to also export properties from the superclass (or some other ancestor class), then a PropertyList can be created which also contains property information from the desired ancestor class. This involves using a different constructor, which takes a second argument specifying the ancestor class from which to copy properties:

```
protected static PropertyList myProps =
   new PropertyList(ThisHost.class, Ancestor.class);

public PropertyInfoList getAllPropertyInfo () {
   return myProps;
}
```

All properties exported by `Ancestor` will now also be exported by `ThisHost`.

What if we want only *some* properties from an ancestor class? In that case, we can edit the `PropertyList` to remove properties we don't want. We can also replace properties with new ones with the same name but possibly different attributes. The latter may be necessary if the class type of a property's value changes in the sub-class:

```
   static
   {
       // remove the property "color"
       myProps.remove ("color");

       // replace the property called "mesh" with one which
       // uses a different kind of mesh object:
       myProps.remove ("mesh");
       myProps.add ("mesh", "quad mesh", null);
   }
```

## 2.5 Composite Properties

A property's value may itself be an object which exports properties; such an object is known as a *composite property*, and its properties are called *sub-properties*.

`Property` handles for sub-properties may be obtained from the top-level exporting class using `getProperty()`, with successive sub-property names delimited by a '`.`' character. For example, if a class exports a property `textureProps`, whose value is a composite property exporting a sub-property called `mode`, then a handle to the mode property can be obtained from the top-level class using

```
   Property mode = getProperty ("textureProps.mode");
```

which has the same effect as

```
   Property texture = getProperty ("textureProps");
   Property mode =
       ((HasProperties)texture).getProperty ("mode");
```

Composite properties should adhere to a couple of rules. First, they should be returned by reference; i.e., the hosting class should return a pointer to the original property, rather than a copy. Secondly, they should implement the `CompositeProperty` interface. This is an extension of `HasProperties` with the following methods:

```
interface CompositeProperty extends HasProperties
{
   // returns the host class exporting this property
   HasProperties getPropertyHost();

   // returns information about this property
   PropertyInfo getPropertyInfo();

   // sets the host class exporting this property
   void setPropertyHost (HasProperties host);

   // sets information for this property
   void setPropertyInfo (PropertyInfo info);
}
```

These methods can be easily implemented using local variables to store the relevant information, as in

```
   HasProperties myHost;

   HasProperties getPropertyHost() {
      return myHost;
   }

   void setPropertyHost (HasProperties host) {
      myHost = host;
   }
```

and similarly for the property information.

The purpose of the `CompositeProperty` interface is to allow traversal of the composite property tree by the property support code.

The accessor method that sets a composite property within a host should set it's host and property information. This can be done using using the `setPropertyHost` and `setPropertyInfo` methods, as in the following example for a compound property of type `TextureProps`:

```
setRenderProps (RenderProps props) {
   if (props != myProps) {
      if (props != null) {
         props.setPropertyInfo(myProps.get("renderProps"));
         props.setPropertyHost(this);
      }
      if (myProps != null) {
         props.setPropertyHost (null);
      }
      myProps = props;
   }
}
```

Alternatively, the same thing can be done using the static convenience method `PropertyUtils.updateCompositeProperty`:

```
setRenderProps (RenderProps props) {
   if (props != myProps) {
      PropertyUtils.updateCompositeProperty (
         this, "textureProps", myProps, props);
      myProps = props;
   }
}
```

## 2.6 Reading and Writing to Persistent Storage

Properties contain built-in support that make it easy to write and read their values to and from persistent storage.

First, `PropertyInfo` contains the methods

```
void writeValue (Object value, PrintWriter pw,
                 NumberFormat fmt);

Object scanValue (ReaderTokenizer rtok);
```

which allow an individual object value to written to a `PrintStream` or scanned from a `ReaderTokenizer`.

Second, if the host object maintains a `PropertyList`, it can use the convenience method

```
void write (
    HasProperties host, PrintWriter pw, NumberFormat fmt);
```

to write out values for all properties for which `getAutoWrite()` returns true. Properties will be written in the form

```
propertyName = value
```

where *value* is the output from the `writeValue` method of the `PropertyInfo` structure.

To economize on file space, there is another method which only writes out property values when those values differ from the property's default value:

```
boolean writeNonDefaults (
    HasProperties host, PrintWriter pw, NumberFormat fmt)
```

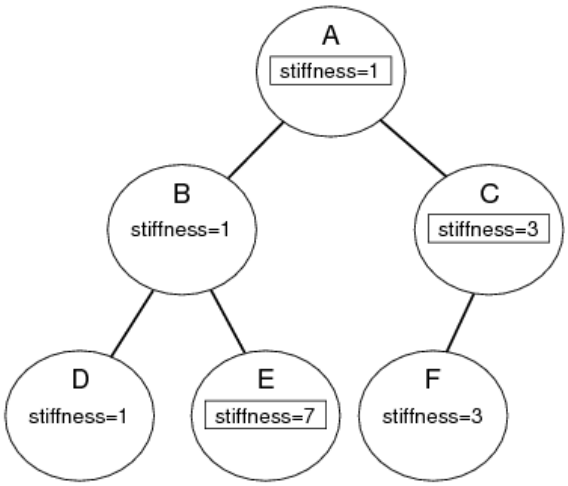


Figure 1: Inheritance of a property named “stiffness” within a hierarchy of property-exporting objects. Explicitly set instances of the property are surrounded by square boxes.

Again, values are written only for the properties for which `getAutoWrite()` returns true. The method returns false if not property values are written.

To read in property values, their are the methods

```
boolean scanProperty (
    HasProperties host, ReaderTokenizer rtok);

boolean scanSpecificProperty (
    HasProperties host, ReaderTokenizer rtok, String name);
```

where the former will inspect the input stream and scan in any recognized property of the form `propertyName = value` (returning true if such a property was found), while the latter will check the input for a property with a specific name (and return true if the specified property was found).

## 2.7 Inheritable Properties

Suppose we have a hierarchical arrangement of property-exporting objects, each exporting an identical property called `stiffness` whose value is a double (properties are considered identical if they have the same name and the same value type). It might then be desirable to have `stiffness` values propagate down to lower nodes in the hierarchy. For example, a higher level node might be a finite element model, with lower nodes corresponding to individual elements, and when we set `stiffness` in the model node, we would like it to propagate to all element nodes for which `stiffness` is not explicitly set. To implement this, each instance of `stiffness` is associated with a *mode*, which may be either *explicit* or *inherited*. When the mode is inherited, `stiffness` obtains its value from the first ancestor object with a `stiffness` property whose mode is explicit.

This is an example of *property inheritance*, as illustrated by Figure 1. Stiffness is explicitly set in the top node (A), and its value of 1 propagates down to nodes B and D whose stiffness mode is inherited. For node C, stiffness is also explicitly set, and its value of 4 propagate down to node F.

Another common use of property inheritance is in setting render properties: we might like some properties, such as color, to propagate down to descendant nodes for which a color has not been explicitly set.

To state things more generally: any property which can be inherited is called an *inheritable* property, and is associated with a mode whose value is either explicit or inherited. The basic operating principle of property inheritance is this:

> **Important:**
> An inherited property’s value should equal that of the nearest matching ancestor property which is explicitly set.

Other behaviors include:

- Setting a property's value (using either the set accessor in the host or the *set* method of the `Property` handle) will cause its mode to be set to *explicit*.
- A property's mode can be set directly. When set to explicit, all descendant nodes in the hierarchy are updated with the property's value. When set to inherited, the property's value is reset from the first explicit value in the ancestry, and then propagated to the descendants.
- When a new node is added to the hierarchy, all inherited properties within the node are updated from the ancestry, and then propagated to the descendants.

If a property is inheritable, then the `isInherited()` method in its `PropertyInfo` structure will return true, and it's property handle will be an instance of `InheritableProperty`:

```
interface InheritableProperty extends Property
{
   // sets the property's mode
   public void setMode (PropertyMode mode);

   // returns the property's mode
   public PropertyMode getMode();
}
```

Valid modes are `PropertyMode.Explicit`, `PropertyMode.Inherited`, and `PropertyMode.Inactive`. The latter is similar to `Inherited`, except that setting an `Inactive` property's value will *not* cause its mode to be set to `Explicit` and its new value will not be propagated to hierarchy descendants.

The hierarchy structure which we have been describing is implemented by having host classes which correspond to hierarchy nodes implement the `HierarchyNode` interface.

```
interface HierarchyNode
{
   // returns an iterator over this node's children
   Iterable<? extends HierarchyNode> getChildren();

   // returns true if this node has children
   boolean hasChildren();

   // returns the parent of this node, if any
   HierarchyNode getParent();
}
```

These methods should be implemented as wrappers to the underlying hierarchy implementation.

## 2.8 Exporting Inheritable Properties

The property package provides most of the code required to make inheritance work, and so all that is required to implement an inheritable property is to provide some simple template code within its exporting class. We will illustrate this with an example.

Suppose we have a property called "width" that is to be made inheritable. Then addition to it's value variable and set/get accessors, the host class should provide a `PropertyMode` variable along with set/get accessors:

```
   int myWidth;
   PropertyMode myWidthMode = PropertyMode.Inherited;

   public PropertyMode getWidthMode() {
      return myWidthMode;
   }
```

```
public void setWidthMode (PropertyMode mode) {
   myWidthMode = PropertyUtils.setModeAndUpdate (
      this, "width", myWidthMode, mode);
}
```

The call to `PropertyUtils.setModeAndUpdate()` inside the `set` method ensures that inherited values within the hierarchy are properly whenever the mode is changed. If the mode is set to `PropertyMode.Explicit`, then the property's value needs to be propagated to any descendent nodes for which it is inherited. If the mode is set to `PropertyMode.Inherited`, then the property's value needs to be obtained from the ancestor nodes, and then also propagated to any descendent nodes for which it is inherited.

As mentioned in the previous section, explicitly setting a property's value using the `set` accessor should cause it's property mode to be set to `Explicit` and the new value to be propagated to hierarchy descendents. This can be accomplished by using `PropertyUtils.propagateValue` within the `set` accessor:

```
public void setWidth (int w) {
   myWidth = w;
   myWidthMode = PropertyUtils.propagateValue (
      this, "width", myValue, myWidthMode);
}
```

The actual creation of an inherited property can be done using the `PropertyList` methods

```
void addInheritable (
   String nameAndMethods, String description,
   Object defaultValue)

void addInheritable (
   String nameAndMethods, String description,
   Object defaultValue, String options)
```

instead of the `add` or `addReadOnly` methods. The `nameAndMethods` argument may now specify up to five method names, corresponding, in order, to the get/set accessors for the property value, the getRange accessor, and the get/set accessors for the property's mode. If any of these are omitted or specified as '`*`', then the system searches for names of the form `getXxx`, `setXxx`, `getXxxRange`, `getXxxNode`, and `setXxxMode`, where `xxx` is the property name.

Finally, the host objects which actually correspond to hierarchy nodes must implement the `HierarchyNode` interface as described in the previous section, *and* any routine which adds a node to the hierarchy must also implement the following code fragment:

```
public void addChild (HierarchyNode node) {
   ... add node to the hierarchy ...
   PropertyUtils.updateInheritedProperties (node);
}
```

This ensures that when a node is added, all property values within and beneath it are made consistent with the inheritance hierarchy.

## 2.9 Inheritable and Composite Properties

Property inheritance is not currently implemented for `CompositeProperty` objects, in order to avoid confusion of the inheritance rules. Suppose a class exports a composite property `A`, which in turn exports an inheritable property `B`. Now suppose that `A` is an inheritable property with its mode is set to `Inherited`. Then the entire structure of A, including the value of B and its mode, is inherited, and it is no longer possible to independently set the value of B, even if its mode is Explicit.

However, the leaf nodes of a composite property tree certainly can be inherited. Suppose a class `ThisHost` exports properties `width`, `order`, and `renderProps`, and that the latter is a composite property exporting `width`, `color`, and `size`. The leafs nodes of the composite property tree exported by `ThisHost` are the properties

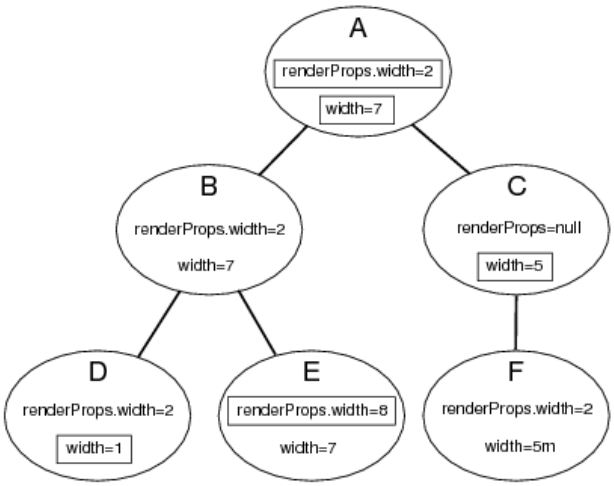


Figure 2: Inheritance of the properties `width` and `renderProps.width` within a hierarchy

```
width
order
renderProps.width
renderProps.color
renderProps.size
```

Each of these may be inheritable, although `renderProps` may not be.

It should be noted that all the leaves in a composite property tree are considered to be unique properties and do not affect each other with respect to inheritance, even if some of the sub-component names are the same. For instance, in the above example, the properties `width` and `renderProps.width` are different; each may inherit, respectively, from occurrences of `width` and `renderProps.width` contained in ancestor nodes, but they do not affect each other. This is illustrated by Figure 2.

Also, if a CompositeProperty is set to `null` within a particular node, then the inheritance of its sub-properties passes straight through that node as though the property was not defined there at all. For example, in Figure 2, `renderProps` is set to `null` in node C, and so `renderProps.width` in node F inherits its value directly from node A.

Composite property inheritance is fully supported if an inheritable property's `set` accessor invokes `PropertyUtils.updateCompositeProperty` as shown in the code example at the end of Section 2.4.2.

# 3 Rendering

The maspack render package supports the graphical rendering of objects using OpenGL and JOGL. An object makes itself renderable by implementing the GLRenderable interface. Renderable objects can then be displayed by a GLViewer, which provides features such as viewpoint control, clipping planes, and component selection.

## 3.1 Renderables and the viewer

Any object to be rendered by GLViewer should implement the GLRenderable interface. Renderables can be added or removed from a viewer using the commands

```
addRenderable (GLRenderable r);
removeRenderale (GLRenderable r);
clearRenderables();
```

To request the viewer to render, one calls the method GLViewer.rerender(), which causes the viewer to redraw itself, using the `render` methods of all its renderables:

```
render (GLRenderer renderer, int flags);
```

This method is called within a rendering thread which usually separate from other application threads (the AWT event thread is commonly used). The renderer interface provides access to JOGL structures, such as `GL2` and `GLU`, which can be used for OpenGL rendering.

```
// render function to draw a single line
void render (GLRenderer renderer, int flags) {
   GL2 gl = renderer.getG2L();
   gl.glDisable (GL2.GL_LIGHTING);
   gl.glLineWidth (2);
   gl.glBegin (GL2.GL_LINES);
   gl.glVertex3f (0, 0, 0);
   gl.glVertex3f (1f, 0, 0);
   gl.glEnd ();
   gl.glLineWidth (1);
}
```

> Note that maspack currently uses JOGL 2, which differs from earlier versions of JOGL in that the class `GL2` replaces the earlier class `GL`.

In addition to providing access to the JOGL GL data structures, the `renderer` argument provides a large number of convenience methods for various graphical operations. The `flags` argument supplies flags that may be used to control different aspects of the rendering. The flags are defined in `GLRenderer`, and adherance to them is recommended but not mandatory. Current flag definitions include:

**SELECTED** Requests that the object be rendered as though it is selected, whether or not it actually is selected;

**VERTEX_COLORING** For meshes, requests that rendering should be done using explicit colors set at the vertices;

**HSV_COLOR_INTERPOLATION** Requests that `HSV` color interpolation should be used when the `VERTEX_COLORING` flag is set;

**SORT_FACES** For polygonal meshes, requests that faces should be sorted in Z direction order. This is to enable better rendering of transparency;

**CLEAR_MESH_DISPLAY_LISTS** For meshes, requests that display lists be cleared.

## 3.2 Prerendering

Because rendering is performed in a thread separate from the main application, this can cause synchronization and consistency problems for renderables which are changing dynamically. For example, suppose we are simulating the motion of two objects, A and B, and we wish to render their positions at a particular time t. If the render thread is allowed to run in parallel with the thread computing the simulation, then A and B might be drawn with positions corresponding to different times (or worse, positions which are indeterminate!). Synchronizing the rendering and simulation threads will aleviate this problem, but that means forgoing the speed improvement of allowing the rendering to run in parallel.

Another option is to give renderables the opportunity to cache their current state for use in the rendering code. This is analagous to double buffering, and can be effected using the prerender() method of GLRenderable:

```
prerender (RenderList list)
```

When rerender is called, the viewer assembles all its renderables into a list. It then iterates through this list, calling `prerender` for each one, within the same thread from which `rerender` was invoked. The renderable can then make a cached copy of any dynamic rendering information, and then use this later when render is called.

The `prerender` method can also be used to add additional renderables to the current renderable list. This is done using the addIfVisible() method of RenderList. For example, if a renderable has two subcomponents, A and B, which it would also like to have rendered, then it can request them to rendered as follows:

```
void prerender (RenderList list) {
   GLRenderable A, B;

   ...
   // add both A and B to the render list
   list.addIfVisible (A);
   list.addIfVisible (B);
}
```

When `addIfVisible` is called, it will check to see if the specified renderable is visible (more on that below), and then, if it is, add it to the list of renderables. In addition, the `prerender` method is recursively called on the specified renderable, whether it is visible or not (since even if a renderable is not visible, it might have subcomponents which are). This allows an entire hierarchy of renderables can be rendered by simply adding the root renderable to the viewer.

Note that any renderables added using the `addIfVisible` method are **not** added to the basic list of viewer renderables specified using addRenderbale and removeRenderable.

Note also that `prerender` should never be called in the rendering thread.

### 3.2.1 Renderable Visibility

As mentioned above, the RenderList.addIfVisible() method only adds renderables to the current render list if they are "visible". Any object implementing GLRenderable is visible by default. However, if the object also implements HasRenderProps, then it is determined to be visible only if the RenderProps returned by HasRenderProps.getRenderProps() is non-null and the associated isVisible method returns true.

## 3.3 Object Selection

GLViewer provides support for the mouse-based selection of renderable components which implement GLSelectable, a subinterface of GLRenderable that implements the following three additional methods

```
boolean isSelectable();

int numSelectionQueriesNeeded();

void getSelection (
   LinkedList<Object> list, int qid);
```

The method isSelectable() should return `true` if the component is in fact selectable. Unless the component manages its own selection behavior (as described in Section 3.3.2), numSelectionQueriesNeeded() should return -1 and getSelection() should do nothing.

Selection is done by identifying all selectables that are completely or partially rendered within a special *selection frustum*, which is a sub-frustum of the current view. Information about these selectables is then passed a `GLSelectionHandler` registered with `GLViewer` (Section 3.4), which then determines the appropriate selection action for each.

*Left-clicking* in the view window will create a selection frustum defined by a 5x5 sub-window centered on the current mouse position. This type of selection is usually handled to produce *single selection* of the most prominant selectable in the frustum.

*Left-dragging* in the view window will create a selection frustum defined by the drag box. This type of selection is usually handleed to produce *multiple selection* of all the selectables in the frustum.

Within GLViewer, selection is implemented in several different ways. If the selection mode requires all objects in the selection frustum, regardless of whether they are clipped by the depth buffer, then OpenGL occlusion queries are used. If only visible objects which have passed the depth test are desired, then a color-based selection scheme is used instead, where each object is rendered with a unique color to an off-screen buffer. Finally, selection may be performed using the (now deprecated) OpenGL `GL_SELECT` rendering mode; to enable this, use the method GLViewer.enableGLSelectSelection().

### 3.3.1 Restrictions when rendering in selection mode

Because color-based selection may be used in the selection process, it is important that application rendering code does not do anything that affects pixel coloring while selection is in progress. In particular, it is important to not:

1. Enable `GL_BLEND`, `GL_LIGHTING`, `GL_TEXTURE`, `GL_FOG`, or `GL_DITHER`;
2. Set colors using `glColor`;

One way to adhear to these restrictions is to conditionalize the relevant calls on whether or not `renderer.isSelecting()` returns true:

```
if (!renderer.isSelecting()) {
   gl.glColor (1f, 0.5f, 0f);
}
```

A more compact option, for colors and lighting control, is to use the following `GLRenderer` methods:

```
setLightingEnabled (boolean enable);
boolean isLightingEnabled();

setTransparencyEnabled (boolean enable);
boolean isTransparencyEnabled();

setColor (float r, float g, float b);
setColor (float r, float g, float b, float a);
setColor (float[] rgbx);
```

These methods will only call the underlying GL primitives if selection is not in progress.

### 3.3.2 Implementing custom selection

By default, if the `isSelectable()` and `numSelectionQueriesNeeded()` methods of a selectable return `true` and `-1`, respectively, then selection will be possible for that component based on whether any portion of it is rendered in the selection frustum. No other programming work needs to be done.

However, in some cases it may be desirable for a selectable to mange it's own selection. A common reason for doing this is that the selectable contains sub-components which are themselves selectable. Another reason might be that only certain parts of what a component renders should be used to indicate selection.

A selectable manages its own selection by adding custom selection code within its `render()` method. This typically consists of surrounding the "selectable" parts of the rendering with *selection queries*, which are indicated by integer identifiers. For example, suppose we have a component which renders in three stages (A, B, and C), and we only want the component to be selected if the rendering for stage A or C appears in the selection frustum. Then we surround the rendering of stages A and C with selection queries, using the GLRenderer methods beginSelectionQuery() and endSelectionQuery():

```
void render (GLRenderer render, int flags) {
   ...
   int qidA = 0;  // selection query for stage A
   int qidC = 1;  // selection query for stage C
   if (renderer.isSelecting()) {
      renderer.beginSelectionQuery (qidA);
   }
   ... render stage A ...
   if (renderer.isSelecting()) {
      renderer.endSelectionQuery ();
   }
   ... render stage B ...
   if (renderer.isSelecting()) {
      renderer.beginSelectionQuery (qidC);
```

```
      }
      ... render stage C ...
      if (renderer.isSelecting()) {
         renderer.endSelectionQuery ();
      }
   }
```

We also need to tell the system how many selection queries we need, and indicate to the system what should be selected in response to a particular query. This is done by creating appropriate declarations for `numSelectionQueriesNeeded()` and `getSelection()`:

```
   int numSelectionQueriesNeeded() {
      return 2;
   }

   void getSelection (LinkedList<Object> list, qid) {
      list.add (this);  // indicate that this component is selected
   }
```

The query index supplied to `renderer.beginSelectionQuery()` should be in the range 0 to `numq`-1, where `numq` is the value returned by `numSelectionQueriesNeeded()`. There is no need to use all requested selection queries, but a given query index should not be used more than once. When rendering associated with a particular query appears in the selection frustum, the system will (later) call `getSelection()` with `qid` set to the query index to determine what exactly has been selected. The selectable answers this by adding the selected component to the `list` argument. Typically only one item (the selected component) is added to the list, but other information can be placed there as well, if an application's selection handler (Section 3.4) is prepared for it.

> A component's `getSelection()` method will be called for each selection query whose associated render fragment appears in the selection frustum. If a component is associated with multiple queries (as in the above example), then its `getSelection()` may be called multiple times.

Note that the use of `beginSelectionQuery(qid)` and `endSelectionQuery()` is conceptually similar to surrounding the render code with `glLoadName(id)` and `glLoadName(-1)`, as is done when implementing selection using the OpenGL `GL_SELECT` rendering mode.

As another example, imagine that a selectable class `Foo` contains a list of selectable components, each of which may be selected individually. The "easy" way to handle this is for `Foo` to hand each component to the `RenderList` in it's `prerender()` method (Section 3.2):

```
   void prerender (RenderList list) {
      for (GLSelectable s : components) {
   list.add (s);
      }
   }
```

Rendering and selection of each component is then handled by the GLViewer.

However, if for some reason (efficiency, perhaps) it is necessary for `Foo` to render the components inside its own `render()` method, then it must also take care of their selection. This can be done by requesting and issuing selection queries for each one:

```
   List<GLSelectable> components; // list of selectable components

   int numSelectionQueriesNeeded() {
      // need one selection query for each component
      return components.size();
   }

   void render (GLRenderer renderer, int flags) {
      int qid = 0; // id for selection query
      for (GLSelectable s : components) {
```

```
if (renderer.isSelecting()) {
   // only render components that are actually selectable ...
         if (renderer.isSelectable(s)) {
            renderer.beginSelectionQuery (qid);
            ... render component ...
            renderer.endSelectionQuery ();
         }
         qid++;
      }
      else {
         ... render component ...
      }
   }
}

void getSelection (LinkedList<Object> list, int qid) {
   // place the selected component onto the list
   list.add (components.get(qid));
}
```

Note that a call to `renderer.isSelectable(s)` is used to determine which selectable components should actually be rendered when a selection render is being performed. This method will return `true` if `s.isSelectable()` returns `true` *and* if `s` is allowed by any selection filters that are currently active in the renderer. Limiting rendering in this way allows components to be selected that might otherwise be hidden by non-selectable components in the foreground.

Finally, what if some of the components in the above example wish to manage their own selection? This can be detected if a component's `numSelectionQueriesNeeded()` method return a non-negative value. In that case, `Foo` can let the component manage its selection by calling its `render()` method, surrounded with calls to beginSubSelection() and endSubSelection(), instead of beginSelectionQuery(int) and endSelectionQuery(), as in

```
void render (GLRenderer renderer, int flags) {
   int qid = 0; // id for selection query
   for (GLSelectable s : components) {
if (renderer.isSelecting()) {
         int numq = s.numSelectionQueriesNeeded();
     if (numq >= 0) {
            // s is managing its own selection
            if (renderer.isSelectable(s)) {
               renderer.beginSubSelection (s, qid);
               s.render (renderer, flags);
               renderer.endSubSelection ();
            }
            // update qid by number of queries requested by s
            qid += numq;
         }
         else {
            if (renderer.isSelectable(s)) {
               renderer.beginSelectionQuery (qid);
               s.render (renderer, flags);
               renderer.endSelectionQuery ();
            }
      qid++;
         }
      }
      else {
         s.render (renderer, flags);
      }
   }
}
```

The call to `beginSubSelection()` sets internal information in the renderer so that *within* the `render()` function for `s`, query indices in the range [0, `numq`-1] correspond to indices in the range [`qid`, `qid`+`numq`-1] as seen outside the render function.

`Foo` must also add the number of selection queries required by its components to the value returned by its own `numSelectionQueries` method:

```
int numSelectionQueriesNeeded() {
   // compute total number of queries required:
   int total = 0;
   for (GLSelectable s : components) {
int numq = s.numSelectionQueriesNeeded();
total += (numq >= 0 ? numq : 1);
   }
   return total;
}
```

Finally, in its `getSelection()` method, `Foo` must delegate to components managing their own selection by calling their own `getSelection()` method. When doing this, it is necessary to offset the query index passed to the component's `getSelection()` method by the base query index for that component, since as indicated above, query indices seen *within* a component are in the range [0, numq-1]:

```
void getSelection (LinkedList<Object> list, int qid) {
   // find component with the matching qid
   int qi = 0;
   for (GLSelectable s : components) {
      int numq = s.numSelectionQueriesNeeded();
      if (numq >= 0) {
   // See if qid is in the range of queries managed by s.
         if (qid >= qi && qid < qi+numq) {
            s.getSelection (list, qid-qi); // offset the query index
            return;
         }
         qi += numq;
      }
      else if (qi == qid) {
         list.add (s);
   return;
      }
   }
}
```

## 3.4 Selection Events

Internally within GLViewer, selection operations are initiated by the method

```
setPick (x, y, width, height, ignoreDepthTest)
```

which sets up selection for all components located within a sub-frustum of the current view defined by a sub-window of dimensions `width` and `height` and centered on `x` and `y`. The flag `ignoreDepthTest` indicates that all components in the frustum should be selected, regardless of whether or not they pass the depth test. (At present, a `true` value for `ignoreDepthTest` causes selection to be performed using occlusion queries instead of color-based selection.)

Components selected by the viewer are indicated to the application via a *selection listener* mechanism, in which the application registers instances of GLSelectionListener with the GLViewer using the methods

```
void addSelectionListener (GLSelectionListener l);
void removeSelectionListener (GLSelectionListener l);
GLSelectionListener[] getSelectionListeners();
```

The listener implements one method with the signature

```
void itemsSelected (GLSelectionEvent e);
```

from which information about the selection can be obtained via a GLSelectionEvent. This provides information about all the queries for which selection occured the methods

```
int numSelectedQueries();
int getFlags();
int getModifiersEx();
LinkedList<Object>[] getSelectedObjects();
```

numSelectedQueries() returns the number of queries that resulted in a selection, getModifiersEx() returns the extended keyboard modifiers that were in play when the selection was requested, and getFlags() returns information flags about the selection (such as whether it was a `DRAG` selection or `MULTIPLE` selection is desired).

Information about the selected components is returned by getSelectedObjects(), which provides an array (of length `numSelectedQueri` of object lists for each selected query. Each object list is the result of the call to getSelection() for that selection query. As indicated in Section 3.3.2, each object list typically contains a single selected component, but may contain other information if the selection handler is prepared for it.

The array provided by `getSelectedObjects()` is ordered so that results for the most visible selectables appear first, so if the handler wishes to select only a single component, it should look at the beginning of the list. Also, if the rendering for a single component is associated with multiple selection queries, mutiple results may returned for that component.

## 3.5 Render Lists and Multiple Viewers

Sometimes, multiple viewers may be used to simultaneously render a common set of renderables. In such cases, it may be wasteful for each viewer to repeatedly execute the prerender phase on the same renderable set. It may also lead to inconsitent results, if the state of renderables changes between different viewers invocation of the prerender phase.

To avoid this problem, an application may execute the prerender phase itself on a set of renderables, and then pass the resulting RenderList to the necessary viewers.

A code sample for this is:

```
GLViewer viewer1;
GLViewer viewer2;
List<GLRenderable> renderables;

...
RenderList rlist = new RenderList();
// execute the pre-render phase
rlist.addIfVisibleAll (renderables);

viewer1.setExternalRenderList (rlist);
viewer2.setExternalRenderList (rlist);

viewer1.rerender();
viewer2.rerender();
```

The render list is initialized to include all visible renderables (see Section 3.2.1, above), and then passed to each viewer by setExternalRenderList(). The contents of this render list are then displayed the next time the viewers redraw themselves, along with any render lists they have generated internally.