Property Expressions

Tapestry uses **property expressions** to move data between components. Property expressions are the basis of the component parameters and template expansions.

A property expression is a string that explains to Tapestry how to navigate from a root object (the containing component) through some number of properties, to a final property that can be read or updated. In fact, that's a simplification, as the properties may actually be public fields, or even method invocations.

As elsewhere in Tapestry, the names of fields, properties, and methods are recognized in a case-insensitive manner.

Related Articles

- Component Templates
- Component Parameters

The most basic form of a property expression is a simple property name, such as "userName".

A series of property names may be specified, separated by periods: "`user.name`", which is equivalent to either `getUser().getName()` or `getUser().setName()`, depending on context (whether the expression is being read or updated).

The "**.**" is called the "dereference operator". In addition, there is a a "safe dereference operator", "**?.**", which works the same as "**.**" except that it allows any of the properties to be null. When reading, a null short-circuits the expression (which returns null). When writing to a property, an intermediate null value will cause the value to be discarded without error.

In other words, "user**?.**name" works, even when the `user` property may be null.

Property expressions can also reference public methods. Methods may take parameters (or not), but must not return void. When the final term in a property expression is a method, then the property expression is read-only.

Being able to invoke methods was originally added so that it was possible to access methods such as `java.util.Map.size()` (which is not named like a property getter method). In Tapestry 5.1, property expressions were extended so that parameters could be passed into methods.

Parameters to methods are, themselves, property expressions. This means that you can write a property expression that reads a property and passes it as a parameter to a method, and then access a property of the object returned from the method.

Compilation

Property expressions are compiled to Java classes at runtime; there is no runtime reflection (unlike the OGNL expression language used in Tapestry 4 and earlier).

Grammar

Expressed in simplified Backus–Naur Form, the grammar of property expressions is as follows:

```
expression : keyword | rangeOp | constant | propertyChain | list | notOp | map;

keyword : 'null' | 'this';

constant : 'true' | 'false' | <integer> | <decimal> | <string>;

rangeOp :  rangeOpArg '..' rangeOpArg;

rangeOpArg : <integer> | propertyChain;

propertyChain : term '.' propertyChain
              | term '?.' propertyChain
              | term;

term :   <identifier>
     |   <identifier> '(' expressionList? ')';

list : '[' expressionList? ']';

expressionList : expression (',' expression)*;

notOpt : '!' expression;

map : '{' '}'
    | '{' mapEntry (',' mapEntry)* '}';

mapEntry : mapKey ':' expression;

mapKey : keyword | constant | propertyChain;
```

Added in 5.3

Support for map literals was added in Tapestry 5.3.

Notes:

- Whitespace is ignored.
- Integers and decimals may have a leading sign ('+' or '-').
- Constants are in base 10 (octal and hex notation is not yet supported). Decimals may contain a decimal point (exponent notation not yet supported).
- Literal strings are enclosed in single quotes.
- The `rangeOp` creates a range object that will iterate between the two values. The upper and lower bounds may be literal integers, or property expressions.
- An identifier by itself is a property name. An identifier with parenthesis is a method invocation.
- Property names, method names, and keywords are case-insensitive.
- 'this' is the root object (i.e., the containing component).
- The `not` operator coerces the expression to a `boolean` (so it can be used on strings, numbers, etc.).
- Method matching is based on method name and number of parameters, but not parameter types. The TypeCoercer service is used to convert parameters to the correct type to be passed into the method.

Examples

	Example	Notes
Keyword	this	
Keyword	null	
Property Name	userName	Calls getUserName() or setUserName, depending on context
Property Chain	user.address.city	Calls getUser().getAddress().getCity() or getUser().getAddress().setCity(), depending on context

Property Chain	user?.name	Calls getUser() and, if the result is not null, calls getName() on the result
Method Invocation	groupList.size()	calls getGroupList().size()
Method Invocation	members.findById(user.id)?.name	Calls getMembers().findById(getUser().getId()).getName() (unless findById returns null)
Range	1..10	Iterates between integers 1 and 10
Range	1..myList.size()	Iterates between 1 and the result of getMyList().size()
Literal String	'Beer is proof that God loves us and wants us to be happy.'	Use single quotes
List	[user.name, user.email, user.phone]	
Not Operator	! user.deleted	the boolean negation of getUser().getDeleted()
Not, Coerced	! user.middleName	true only if getUser.getMiddleName() returns null or an empty string
Map	{ 'framework' : 'Tapestry', 'version' : version }	Keys are string literals (in single quotes), but could be properties as well