

# Table of Contents

Using Properties .....	1
------------------------	---

## Using Properties

A *Property* is an attribute of a module component (or the module itself), simply consisting of a name and a value.

Properties are used by VASSAL when executing commands and for determining details of the current game state or components.

Using Properties is an important part of automating your module.

Properties do nothing by themselves. They need to be evaluated by other VASSAL functions or components. Properties can be used as selection criteria for certain pieces, to track game quantities or game events, and for many other purposes.

### *\*Types of Properties\**

Properties come in two types: system Properties, and custom Properties.

- Many VASSAL components, such as Game Pieces, Decks, Maps, Boards, Dice Rollers, and Turn Counters, already have Properties defined for them. These are called *System Properties*. System Properties for components are listed in the component descriptions in later sections.
- In addition, you can create and define *custom* Properties for Game Pieces, Map Windows, Zones, and the module itself. Examples of custom Properties include Global Properties, Dynamic Property Traits, and Marker Traits. Information on using component Properties, Traits, or Global Properties will be found in their descriptions in this Guide.

### *\*Property Names\**

**System Property Names:** The names of System Properties are already defined for module components. For example, all Game Pieces have a system Property named `CurrentMap`, the value of which is the name the Map where the Game Piece is currently located. System Properties cannot be renamed. System properties for module components are defined in this guide in the sections describing those components.

**Custom Property Names:** The name of a custom Property you define, such as a Global Property, Dynamic Property Trait, or Marker Trait, must be composed of alphanumeric characters (A- Z, 1-9), and may contain a space ( ). The name cannot contain special characters or punctuation marks. To avoid unpredictable behavior, the name of a custom property should not duplicate the name of a System Property.

Property names are case-sensitive. For example, `PowerLevel` is not the same Property as `powerLevel`.

### *\*Property Values\**

**System Property Values:** System Properties automatically take their values from the game state. You won't need to assign values to System Properties manually (in fact, you cannot do so). Their values will depend on the condition they describe. For example, the value of the CurrentBoard Property for a Game Piece is the name of the current Board where the piece is located. If you moved the piece to a new Board, the value of CurrentBoard for that piece would change automatically to the name of the new Board.

**Custom Property Values:** You may assign values to custom Properties. You can assign any of the following types of values:

- *A string of text:* By default, and unless defined otherwise, Properties will accept a string of text as a value. Unless noted otherwise, this is the default type for most Properties. The value of a text Property can include a space ( ) character (for example, *Foo Bar*).
- *Boolean value:* Some Properties are simply checked to see if they are logically true or not. (These are called Booleans.) A Boolean Property will have a text value that can be either "yes/no" or "true/false" (not capitalized).
- *Numerical:* The value of a numerical Property is limited to positive or negative integers (or zero), such as -5, 1, 0, -23, 134, and so on. You can designate some Properties (such as Global and Dynamic Properties) as numerical when you create them.

As with Property Names, Property values are case-sensitive.

*\*Displaying Properties:* Game Piece property values are generally invisible to players unless you choose to display them, using a Trait such as Text Label or Layer.

## Comparing Properties

You create Property *expressions* to determine if a particular game condition is true. For example, since a System Property named LocationName is used to record a piece's current location, we could check the value of this Property to determine if the piece is currently situated in Hex 1212.

*\*Using Properties: Comparing Properties\**

Property expressions are found in many components of a module, and you set them in the dialog boxes for those components when you choose the component settings. Expressions are also sometimes called *filters*, because they filter out situations where the comparison does not apply.

An expression must use one of the following symbols (known as operators) to evaluate the relationship between two values.

*Traditionally, a space character ( ) is placed between the Property, the operator, and the value, to improve legibility. However, spaces between them is not required. (CurrentTurn >= 5 is the same expression as CurrentTurn>=5.)*

The following table shows each valid operator, the meaning, and under what conditions a comparison using the operator will be true.

Operator	Meaning	True if...	

=	Equals	The two values on either side are the same.	
!=	Is Not Equal To	The two values are not the same. (The exclamation point (!) is known as a "bang".)	
=~	Regular Expression	The Property value is equal to any one of several values, which are separated by a	
		pipe (	) character. See <i>Regular Expressions</i> , below, for more information.
	!~	Regular Expression (Negation)	The Property value is <i>not</i> equal to any one of several values, which are separated
			by a pipe (
) character. See <i>Regular Expressions</i> , below, for more information.			
		>	Greater Than
The value on the left side is larger than the value on the right. Applies to Numerical			
Properties only.			
		⇒	Greater Than Or Equal To

The value on the left side is larger than or equal to the value on the right. Applies to			
Numerical Properties only.		<	Less Than
The value on the left side is smaller than the value on the right. Applies to			
Numerical Properties only.		≤	Less Than Or Equal To
The value on the left side is smaller than or equal to the value on the right. Applies			

### *\*Types of Expressions\**

Typically, expressions are used in a module component to determine the conditions under which the effects of the component should apply. There are several kinds of expressions, which include:

- Simple expressions, which check the Property to see if matches a single value.
- Regular expressions, which check the Property for any of several values.
- Comparing the value of the Property to the value of another Property.
- Indirect comparisons, where one Property name contains the name of another Property.
- Joined comparisons, which can check for multiple conditions.

*When creating comparisons, remember that Property names and values are case-sensitive.*

### **Simple Expressions**

To check if the value of a Property matches a single value, use a simple expression. For example:

- PieceName = Paratrooper (text)
- CurrentTurn ⇒ 10 (numerical)
- ObscuredToOthers = true (Boolean)

*In these comparisons, the value on the right side is called a literal, because the text, number, or condition must be literally true—as written—for the comparison to be true.*

### **Regular Expressions**

A *regular expression* checks if a Property has any one of several values. A regular expression is denoted using the `=~` operator. Surround the name of the Property on the left side with `$`-signs, and separate each value by a pipe character (`|`). There must be no spaces between pipe-separated values. For example:

□ `CurrentPlayer =~ Blue | Green | Red` (checks if the Blue, Green or Red player is the current player)

*\*Using Properties: Game Piece Properties\**

You can also negate regular expressions by using `!~` instead of `=~`.

## Comparing a Property to Another Property

On occasion, you may need to compare the value of one Property to the value of another. In this case, surround the name of the Property on the right side of the operator with `$`-signs (such as `$PieceName$`) to indicate that the Property with that name should be checked for its value. (Do not use `$`-signs in the left side of the expression. The left side of the expression is always treated as the name of a Property.) Examples:

- `PieceName = $ActivePiece$` (checks if the name of a selected piece is the same as the value of the `$ActivePiece$` Global Property.)
- `CurrentTurn = $2d6_result$` (checks if the current turn is the same as the random roll of 2 dice.)

*In these comparisons, the Property on the right, in `$`-signs, is called a variable, because its value may vary.*

## Indirect Comparisons

In an indirect comparison, one Property name contains the value of another Property. Set the name of the Property in the left side by using `$`-signs. For example, if the Property `Example` has a Property name as a value, then to compare the value of the Property contained in `Example` to a value, use `$` on the left side of the operator.

- `$Example$ = 2`

Use `$` (dollar) signs within the name of a custom Property to indicate that the Property contains the name of another Property. For example, in a game with Red, Green and Blue players, the value of the `$PlayerSide$` Property can be *Red*, *Green*, or *Blue*. Using the Send to Location Trait, we want to send a card to the current active player's private window (each named `Red_Home`, `Green_Home`, `Blue_Home`). For the Trait's destination we could use the Property `$PlayerSide$_Home`. When evaluated, the value of `$PlayerSide$` would be substituted in the string, giving a final value for `$PlayerSide$_Home` of `Red_Home`, `Green_Home`, or `Blue_Home`.

## Joining Expressions

You can check for multiple conditions using AND (`&&`) as well as OR (`|`) to join expressions together. For example, to check if a

Game Piece's current board was called *Battlefield*, *and* that the piece was an Artillery piece, we would evaluate:

CurrentBoard = Battlefield && PieceName = Artillery

- In an AND comparison, both compared Properties must be true for the entire expression to be true.
- In an OR comparison, only one of the compared Properties must be true for the entire expression to be true.

Complex expressions with multiple joins are possible. (Parentheses and brackets are not supported.) Joined expressions are evaluated from left to right, with OR (|) operators evaluated before AND (&&).

For example,

CurrentBoard = HQ || CurrentBoard = Battlefield && PieceName = Artillery || PieceName = Tank

This would evaluate to *true* if the piece were on either the HQ or Battlefield maps, and was either an Artillery or Tank unit. If the piece were on the HQ map, but was an infantry unit, it would evaluate to *false*.

## Game Piece Properties

Each Game Piece has its own set of System Properties (each with a name and a value) that can be used for identification by various components.

When looking for the value of a Property of a Game Piece, Global Properties provide the default values. If the Property is not defined on the Game Piece itself, the value will come from a Global Property attached to Zone occupied by piece, the Map to which it belongs, or the Module overall, in that order.

Traits on a Game Piece search for Properties in the following order:

1. Within each Trait on itself in order from the Trait at the bottom of the list, up to the top Trait.
2. Zone Global Properties defined for the Zone where the Game Piece is currently located.
3. Map Global Properties defined for the Map where the Game Piece is currently located.
4. Global Properties defined at the module level.

A Game Piece cannot directly access:

*\*Using Properties: Message Formats\**

- Properties on another Game Piece.
- Zone Global Properties on a Zone that the Game Piece is not currently located in.
- Map Global Properties on a map that the Game Piece is not currently located in.

For most components, system Properties are hardcoded as part of the VASSAL engine. However, for Game Pieces, you can create entirely new Properties using the Dynamic Property, Marker, and Property Sheet Traits. See *Game Piece Traits* on page 42 for more information.

## Message Formats

Many Traits and module components enable you to customize the message that is displayed to users in the Chat Window when game events take place. A *Message Format* is a formula for creating such a message to players. Message formats are highly customizable and usually include Properties as variables.

For example, the Dice button control includes a message indicating the result of the dice, which is specified in **Report Format**. The default message for the Dice button is **\$name\$ = \$result\$\*<\$playerName\$>**. This formula indicates the format of the message to be displayed.

- \$name\$ is evaluated for the name of the Dice button.
- \$result\$ is the results of the roll.
- \$playerName\$ is the name of the player who clicked the button.

If Bill clicked a Dice button named 2d6, and the result was 5, the message displayed in the Chat Window would be: **2d6 = 5\*<Bill>**.

#### *\*Constructing a Message Format\**

In a Message Format, any word surrounded by \$-signs represents a variable, the value of which will be determined when the message is generated during play. When constructing a Message Format for a component, click the **Insert** drop-down menu for a list of available variables for the Message Format. Selecting one of the variables from the menu will insert it at the current cursor position.

Words not surrounded by \$-signs will be treated as plain text. This enables you to create plain-language messages using a combination of text and variables.

When a Message Format is used in conjunction with a Game Piece, then any Properties of that Game Piece can be used in the Message Format. See page 44 for more information on Game Piece Properties.