Gosu Reference Guide

ClaimCenter Release 6.0.8

Product Name: Guidewire ClaimCenter
Product Release: 6.0.8
Document Name: Gosu Reference Guide
Document Revision: 05-February-2013

Contents

About This Document

This document is a guide for the syntax of Gosu expressions and statements. It also provides examples of how the syntax can be constructed to write scripts (for example, in rules, libraries, and user interface elements).

Intended Audience

This document is intended for the following readers:

- Programmers who implement and maintain Gosu code
- Implementation team members who configure any part of an application that invokes Gosu. For example, the rules and web user interface configuration.
- Implementation team members or Information Technology staff who do not directly write Gosu rules, but seek a better understanding of how Guidewire applications use rules to perform business logic

Assumed Knowledge

This document assumes that you are familiar with using Guidewire Studio. For an overview of Guidewire Studio, see the *ClaimCenter Configuration Guide*.

Related Documents

See the following Guidewire documents for further information:

ClaimCenter Application Guide – Introduces the application, explains application concepts, and provides a high-level view of major features and business goals of ClaimCenter. This is your first place to look when learning about a feature. This book is written for all audiences.

ClaimCenter Data Dictionary – Describes the ClaimCenter data model, including your custom data model extensions. To generate the dictionary, go to the `ClaimCenter/bin` directory and run the `gwcc regen-dictionary` command. To view the dictionary, open the `ClaimCenter/build/dictionary/data/index.html` file. For more information about generating and using the *Data Dictionary*, see the *ClaimCenter Configuration Guide*.

ClaimCenter Rules Guide – Describes the business rule methodology, rule categories for ClaimCenter, and rule syntax for Guidewire Studio. This book is intended for programmers who write Gosu business rules and analysts who define the business rule logic.

Conventions In This Document

Text style	Meaning	Examples
italic	Emphasis, special terminology, or a book title.	A *destination* sends messages to an external system.
bold	Strong emphasis within standard text or table text.	You **must** define this property.
narrow bold	The name of a user interface element, such as a button name, a menu item name, or a tab name.	Next, click **Submit**.
`monospaced`	Literal text that you can type into code, computer output, class names, URLs, code examples, parameter names, string literals, and other objects that might appear in programming code.	Get the field from the `Address` object.
`monospaced italic`	Parameter names or other variable placeholder text within URLs or other code snippets.	Use `getName(`*`first`*`, `*`last`*`)`. `http://`*`SERVERNAME`*`/a.html`.

chapter 1

Gosu Introduction

This topic introduces the Gosu language, including basic syntax and a list of features.

This topic includes:

Welcome to Gosu

Welcome to the Gosu language. Gosu is a general-purpose programming language built on top of the Java Virtual Machine. It includes the following features:

- *object-oriented*
- *easy to learn*, especially for programmers familiar with Java
- *static typing*
- *imperative*
- *Java compatible* (use Java types, extend Java types, implement Java interfaces)
- *type inference* greatly simplifies your code while still preserving static typing
- *blocks*, which are in-line functions that you can pass around as objects. Some languages call these closures or lambda expressions.

- *enhancements*, which add functions and properties to other types, even Java types. Gosu includes built-in enhancements to common Java classes, some of which add features that are unavailable in Java (such as blocks).
- *generics*, which abstracts the behavior of a type to work with multiple types of objects. The Gosu generics implementation is 100% compatible with Java, and adds additional powerful improvements. See "Generics in Gosu" on page 30 for details.
- *native XML and XSD support*, which Java Gosu includes as an included type loader
- native SOAP support
- large companies all around the world use Gosu every day in their production systems for critical systems

Basic Gosu

The following Gosu program outputs the text `"Hello World"` to the console using the built-in `print` function:

```
print("Hello World")
```

As you can tell, the delimiters around text literals are double quotes. Gosu uses the Java type `java.util.String` as its native `String` type. You can create in-line `String` literals just as in Java. In addition, Gosu supports native in-line templates, which simplifies common text substitution coding patterns. For more information, see "Gosu Templates" on page 25.

To declare a variable in the simplest way, use the `var` statement followed by the variable name. Typical Gosu code also initializes the variable using the equals sign followed by any Gosu expression:

```
var x = 10
var y = x + x
```

Despite appearances in this example, Gosu is *statically typed*. All variables have a compile-time type that Gosu enforces at compile time, even though in this example there is no *explicit* type declaration. In this example, Gosu automatically assigns these variables the type `int`. Gosu *infers* the type `int` from the expressions on the right side of the equals signs on lines that declare the variable. This language feature is called *type inference*. For more information about type inference, see "Type Inference" on page 28.

Type inference helps keep Gosu code clean and simple, especially compared to other statically-typed programming languages. This makes typical Gosu code easy to read but retains the power and safety of static typing. For example, take the common pattern of declaring a variable and instantiating an object.

In Gosu, this looks like:

```
var c = new MyVeryLongClassName()
```

This is equivalent to the following Java code:

```
MyVeryLongClassName c = new MyVeryLongClassName();
```

As you can see, the Gosu version is easier to read and more concise.

Gosu also supports **explicit** type declarations of variables during declaration by adding a colon character and a type name. The type name could be a language primitive, a class name, or interface name. For example:

```
var x : int = 3
```

Explicit type declarations are required if you are **not** initializing the variable on the same statement as the variable declaration. Explicit type declarations are also required for all class variable declarations.

> **Note:** For more information, see "More About the Gosu Type System" on page 27 and "Gosu Classes and Properties" on page 17.

From the previous examples, you might notice another difference between Gosu and Java: no semicolons or other line ending characters. Semicolons are unnecessary in nearly every case, and the standard style is to omit them. For details, see "Gosu Statement Terminators" on page 33.

Control Flow

Gosu has all the common control flow structures, including improvements on the Java versions.

Gosu has the familiar `if`, `else if`, and `else` statements:

```
if( myRecord.Open and myRecord.MyChildList.length > 10 ) {
   //some logic
 } else if( not myRecord.Open ) {
   //some more logic
 } else {
   //yet more logic
 }
```

Gosu permits the more readable English words for the Boolean operators: `and`, `or`, and `not`. Optionally you can use the symbolic versions from Java (`&&`, `||`, and `!`). This makes typical control flow code easier to understand.

The `for` loop in Gosu is similar to the Java 1.5 syntax:

```
for( ad in addressList ) {
   print( ad.Id )
 }
```

This works with arrays or any `Iterable` object. Despite appearances, the variable is strongly typed. Gosu infers the type based on the iterated variable's type. In the previous example, if `addressList` has type `Address[]`, then `ad` has type `Address`. If the `addressList` variable is `null`, the `for` statement is skipped entirely, and Gosu generates no error. In contrast, Java throws an `null` pointer exception if the iterable object is `null`.

If you want an index within the loop, use the following syntax to access the zero-based index:

```
for( a in addressList index i) {
   print( a.Id + " has index " + i)
}
```

Note: Gosu does not have a direct equivalent of the Java three-part `for` declaration:

```
for ( i =1 ; i <20 ; ++i )
```

The Gosu `switch` statement can test any type of object, with a special default case at the end:

```
var x = "b"

switch( x ) {
  case "a":
    print("a")
    break
  case "b":
    print("b")
    break
  default:
    print("c")
}
```

In Gosu, you must put a `break` statement at the end of each case to jump to the end of the `switch` statement. Otherwise, Gosu falls through to the next case in the series. For example, for the previous example if you **remove** the `break` statements, the code prints both `"b"` and `"c"`. This is the same as Java, although some languages do not require the `break` statement to prevent falling through to the next case.

Blocks

Gosu supports in-line functions that you can pass around as objects. Some languages call these *closures* or *lambda expressions*. In Gosu, these are called *blocks*.

To define a block

1. start with the \ character
2. optionally add a list of arguments as name/type pairs separated by a colon character
3. add the `->` characters, which mark the beginning of the block's body
4. finally, add either a statement list surrounded by curly braces: `{` and `}`, or a Gosu expression.

For more information about blocks, see "Gosu Blocks" on page 213.

The following block multiplies a number with itself, which is known as squaring a number:

```
var square = \ x : Number-> x * x    / /no need for braces here (it is an expression, not statements)
var myResult = square(10)         // call the block
```

The value of `myResult` in this example is 100.

Blocks are incredibly useful as method parameters, which allows the method's implementation to generalize some task or algorithm but allow callers to inject code to customize it. For example, Gosu adds many useful methods to Java collections classes that take a block as a parameter. That block could return an expression (for example, a condition to test each item against) or could represent an action to perform on each item.

For example, the following Gosu code makes a list of strings, sorts it by length of each `String`, then iterates across the result list to print each item in order:

```
var strings = {"aa", "ddddd", "c"}
strings.sortBy( \ str -> str.Length ).each( \ str -> { print( str ) } )
```

For more information about blocks, see "Gosu Blocks" on page 213. For more information about collections enhancement methods, many of which use blocks, see "Collections" on page 231.

Special Block Shortcut for One-Method Interfaces

If the anonymous inner class implements an interface and the interface has **exactly one method**, then you can use a Gosu block to implement the interface as a block. This is an alternative to using an explicit anonymous class. This is true for interfaces originally implemented in either Gosu or Java. For example:

```
_callbackHandler.execute(\ -> { /* your Gosu statements here */ })
```

For more information, see "Gosu Block Shortcut for Anonymous Inner Classes Implementing an Interface" on page 186.

Enhancements

Gosu provides a feature called *enhancements*, which allow you to add functions (methods) and properties to other types. This is especially powerful for enhancing native Java types, and types defined in other people's code.

For example, Gosu includes built-in enhancements on collection classes (such as `java.util.List`) that significantly improve the power and readability of collections-related code. For example, the example mentioned earlier takes a list of `String` objects, sorts it by length of each `String`, and iterates across the result list to print each item:

```
strings.sortBy( \ str -> str.Length ).each( \ str -> print( str ) )
```

This works because the `sortBy` and `each` methods are Gosu enhancement methods on the `List` class. Both methods return the result list, which makes them useful for chaining in series like this.

Collections

Gosu provides several features to make it easy to use collections like lists and maps. Gosu directly uses the built-in Java collection classes like `java.util.ArrayList` and `java.util.Hashmap`. This makes it especially easy to use Gosu to interact with pre-existing Java classes and libraries.

In addition, Gosu adds the following features:

- Shorthand syntax for creating lists and maps that is easy to read and still uses static typing:

```
var myList = {"aa", "bb}
var myMap = {"a" -> "b", "c" -> "d"}
```

- Shorthand syntax for getting and setting elements of lists and maps

```
var myList = {"aa", "bb}
myList[0] = "cc"
var myMap = {"a" -> "b", "c" -> "d"}
var mappedToC = myMap["c"]
```

- Gosu includes built-in enhancements that improve Java collection classes. Some enhancements enable you to use Gosu features that are unavailable in Java. For example, the following Gosu code initializes a list of

String objects and then uses enhancement methods that use Gosu blocks, which are in-line functions. (See “Blocks” on page 15).

```
// use Gosu shortcut to create a list of type ArrayList<String>
var myStrings = {"a", "abcd", "ab", "abc"}

// Sort the list by the length of the String values:
var resortedStrings = myStrings.sortBy( \ str -> str.Length )

// iterate across the list and run arbitrary code for each item:
resortedStrings.each( \ str -> print( str ) )
```

Notice how the collection APIs are chainable. For readability, you can also put each step on separate lines. The following example declares some data, then searches for a subset of the items using a block, and then sorts the results.

```
var minLength = 4
var strings = { "yellow", "red", "blue" }

var sorted = strings.where( \ s -> s.length() >= minLength )
                    .sort()
```

For more information, see “Collections” on page 16.

Access to Java Types

Gosu provides full access to Java types from Gosu. You can continue to use your favorite Java classes or libraries directly from Gosu with the same syntax as native Gosu objects.

For example, for standard Gosu coding with lists of objects, use the Java type `java.util.ArrayList`. The following is a simple example using a Java-like syntax:

```
var list = new java.util.ArrayList()
list.add("Hello Java, from Gosu")
```

For example:

- Gosu can instantiate Java types
- Gosu can manipulate Java objects (and primitives) as native Gosu objects.
- Gosu can get variables from Java types
- Gosu can call methods on Java types. For methods that look like getters and setters, Gosu exposes methods instead as properties.
- Gosu extends and improves many common Java types using Gosu *enhancements*. (See “Enhancements” on page 16.)
- You can also extend Java types and implement Java interfaces.

For more information, see “Java and Gosu” on page 101.

Gosu Classes and Properties

Gosu supports object-oriented programming using classes, interfaces and polymorphism. Also, Gosu is fully compatible with Java types, so Gosu types can extend Java types, or implement Java interfaces.

At the top of a class file, use the `package` keyword to declare the *package* (namespace) of this class. To import specific classes or package hierarchies for later use in the file, add lines with the `uses` keyword. This is equivalent to the Java `import` statement. Gosu supports exact type names, or hierarchies with the `*` wildcard symbol:

```
uses gw.example.MyClass          // exact type
uses gw.example.queues.jms.*     // wildcard means a hierarchy
```

To create a class, use the `class` keyword, followed by the class name, and then define the variables, then the methods for the class. To define one or more constructor (object instance initialization) methods, use the `construct` keyword. The following is a simple class with one constructor that requires a `String` argument:

```
class ABC {
  construct( id : String ) {
  }
```

```
}
```

Note: You can optionally specify that your class implements interfaces. See “Interfaces” on page 21.

To create a new instance of a class, use the `new` keyword in the same way as in Java. Pass any constructor arguments in parentheses. Gosu decides what version of the class constructor to use based on the number and types of the arguments. For example, the following calls the constructor for the ABC class defined earlier in this topic:

```
var a = new ABC("my initialization string")
```

Gosu improves on this basic pattern and introduces a standard compact syntax for *property initialization* during object creation. For example, suppose you have the following Gosu code:

```
var myFileContainer = new my.company.FileContainer()
myFileContainer.DestFile = jarFile
myFileContainer.BaseDir = dir
myFileContainer.Update = true
myFileContainer.WhenManifestOnly = ScriptEnvironment.WHEN_EMPTY_SKIP
```

After the first line, there are four more lines, which contain repeated information (the object variable name).

You can optionally use Gosu object initializers to simplify this code to only a couple lines of code:

```
var myFileContainer = new my.company.FileContainer() { :DestFile = jarFile, :BaseDir = dir,
  :Update = true, :WhenManifestOnly = ScriptEnvironment.WHEN_EMPTY_SKIP }
```

You can also choose to list each initialization on its own line, which takes up more lines but is more readable:

```
var myFileContainer = new my.company.FileContainer() {
  :DestFile = jarFile,
  :BaseDir = dir,
  :Update = true,
  :WhenManifestOnly = ScriptEnvironment.WHEN_EMPTY_SKIP
}
```

For more details, see “Creating and Instantiating Classes” on page 170.

Functions

Declare a function using the `function` keyword. When a function is part of another type, a function is called a *method*. In Gosu, types follow the variable or function definition, separated by a colon. In contrast, Java types precede the variable or parameter name with no delimiter. To return a value, add a statement with the `return` keyword followed by the value. The following simple function returns a value:

```
public function createReport( user : User) : Boolean {
  return ReportUtils.newReport(user, true)
 }
```

Method invocation in Gosu looks familiar to programmers of imperative languages, particularly Java. Just use the period symbol followed by the method name and the argument list in parentheses:

```
obj.createReport( myUser )
```

Pass multiple parameters (including Gosu expressions) delimited by commas, just as in Java:

```
obj.calculate(1, t.Height + t.Width + t.Depth)
```

In some cases, such as in-line functions in Gosu programs, functions are not attached to a class or other type. In such cases, simply call them. As you saw in earlier examples, there is a rare globally-available function for any Gosu code, called `print`. Call that function with a `String` to write data to the system console or other default output stream. For example, the following prints text to the console:

```
print("Hello Gosu!")
```

Gosu supports access control modifiers (`public`, `private`, `internal`, and `protected`) and they have the same meaning as in Java. For example, if you mark a method `public`, any other code can call that method. For more information, see “Access Modifiers” on page 178.

Gosu also supports static methods, which means methods on the type rather than on object instances. See “Static Members” on page 20.

If the return type is not `void`, **all** possible code paths must return a value in a method that declares a return type. In other words, if any code path contains a `return` statement, Gosu requires a `return` statement for all possible paths through the function. The set of all paths includes all outcomes of conditional execution, such as `if` and `switch` statements. This is identical to the analogous requirement in Java. The Gosu editor automatically notifies you at compile time of this issue if it happens. For details, see “Gosu Functions” on page 90.

Class Variables and Properties

Gosu supports instance variables and static variables in class declarations in basically the same way Java does, although the syntax is slightly different. Use the `var` keyword in the class definition, and declare the type explicitly. Note that variables are `private` by default in Gosu.

```
var _id : String            //vars are private by default
```

One special difference between Gosu and some languages (including Java) is full support in Gosu for **properties**, which are dynamic getter and setter methods for values. To set or get properties from an object (internally, Gosu calls the property getter and setter methods), use natural syntax. Type the period (.) character followed by the property name just as you would for an object variable:

```
var s = myobj.Name
myobj.Name = "John"
```

In addition, Gosu has a special null-safety behavior with pure property paths, which are the form `obj.Property1.Property2.Property3`. For more information, see “Property Accessor Paths are Null Safe” on page 20.

Define a property accessor function (a property getter) using the declaration `property get` instead of `function`. Define a setter function using function declaration `property set` instead of `function`. These property accessors can dynamically get or set the property, depending on whether it is defined as `property get` or `property set`. Properties can be read/write, or can be read-only or write-only. Gosu provides a special shortcut to expose internal variables as public properties with other names. Use the syntax `as` *`PROPERTY_NAME`* as follows in a class definition for a variable. This makes it easy to separate the internal implementation of variables from how you expose properties to other code

```
var _name : String as Name  //Exposes the _name field as a readable and writable 'Name' property
```

Think of this is a shortcut for creating a `property get` function and a `property set` function for each variable. This is the standard and recommended Gosu style for designing public properties. (In contrast, for new Gosu code do not expose actual class variables as public, although Gosu supports it for compatibility with Java.)

The following is a simple Gosu class definition:

```
package example             // declares the package (namespace) of this class

uses java.util.*            // imports the java.util package

 class Person {

   var _name : String as Name  // Exposes the _name field as a readable and writable 'Name' property
   var _id : String            // vars are private by default

  //Constructors are like functions called construct but omit the function keyword.
  // You can supply multiple method signatures with different numbers or types of arguments
    construct( id : String ){
     _id = id
   }

   property get ID() : String {         //_id is exposed as a read only 'ID' property
     return _id
   }

   // Comment out the property set function to make ID read-only property:
   property set ID(id : String) {
    _id = id;
   }

   //functions by default are public
   function printOut() {
     print(_name + ":" + _id)
   }
 }
```

This allows you to use concise code like the following:

```
n.ID = "12345"     // set a property
print(n.ID)        // get a property
n.Name = "John"    // set a property -- see the "as Name" part of the class definition!
print(n.Name)      // get a property -- see the "as Name" part of the class definition!
```

From Gosu, Java Get and Set Methods Become Properties

For methods on Java types that look like getters and setters, Gosu exposes methods on the type as properties rather than methods. Gosu uses the following rules for methods on Java types:

- If the method name starts with `set` and takes exactly one argument, Gosu exposes this as a property. The property name matches the original method but without the prefix `set`. For example, suppose the Java method signature is `setName(String thename)`. Gosu exposes this a property set function for the property called `Name` of type `String`.
- If the method name starts with `get` and takes no arguments and returns a value, Gosu exposes this as a getter for the property. The property name matches the original method but without the prefix `get`. For example, suppose the Java method signature is `getName()` and it returns a `String`. Gosu exposes this a `property get` function for the property named `Name` of type `String`.
- Similar to the rules for `get`, the method name starts with `is` and takes no arguments and returns a Boolean value, Gosu exposes this as a property accessor (a *getter*). The property name matches the original method but without the prefix `is`. For example, suppose the Java method signature is `isVisible()`. Gosu exposes this a `property get` function for the property named `Visible`.

If there is a setter and a getter, Gosu makes the property readable and writable. If the setter is absent, Gosu makes the property read-only. If the getter is absent, Gosu makes the property write-only.

For example, consider a Java class called `Circle` with the following method declarations:

```
public double getRadius()
//...
public void setRadius(double dRadius)
```

Gosu exposes these methods as the `Radius` property, which is readable and writable. That means you could use straightforward code such as:

```
circle.Radius = 5 // property SET
print(circle.Radius) // property GET
```

For a detailed example, see “Java Get and Set Methods Convert to Gosu Properties” on page 104.

Property Accessor Paths are Null Safe

One notable difference between Gosu and some other languages is that property accessor paths in Gosu are *null tolerant*, also called *null safe*. This affects only expressions separated by period characters that access a series of *instance variables* or *properties* such as:

```
obj.Property1.Property2.Property3
```

In most cases, if any object to the left of the period character is `null`, Gosu does not throw a *null pointer exception* (NPE) and the expression returns `null`. Gosu null-safe property paths tends to simplify real-world code. Gosu null-tolerant property accessor paths are a very good reason to expose data as *properties* in Gosu classes and interfaces rather than as setter and getter methods.

For details of this important Gosu feature, see “Property Accessor Paths are Null Safe” on page 29.

Static Members

Gosu supports *static* members on a type. This includes variables, functions, property declarations, and static inner classes on a type. The static quality means that the member exists only on the *type* (which exists only once), not on *instances* of the type. The syntax is simple. After a type reference (just the type name), use the period (.) character followed by the property name or method call. For example:

```
MyClass.PropertyName // get a static property name
MyClass.methodName() // call a static method
```

In Gosu, for each usage of a static member you must *qualify* the class that declares the static member. However, you do *not* need to fully-qualify the type. In other words, you do not need to include the full package name if the type is already imported with a `uses` statement or is already in scope. For example, to use the `Math` class's cosine function and its static reference to value `PI`, use the syntax:

```
Math.cos(Math.PI * 0.5)
```

Gosu does not have an equivalent of the *static import* feature of Java 1.5, which allows you to omit the enclosing type name before static members. In the previous example, this means omitting the text `Math` and the following period symbol. This is only a syntax difference for using static members in Gosu code, independent of whether the type you want to import is a native Gosu or Java type.

To declare a type as static for a new Gosu class, use the `static` keyword just as in Java. For details, see "Modifiers" on page 178.

Interfaces

Gosu supports interfaces, including full support for Java interfaces. An interface is a set of method signatures that a type must implement. It is like a contract that specifies the minimum set of functionality to be considered compatible. To implement an interface, use the `interface` keyword, then the interface name, and then a set of method signatures without function bodies. The following is a simple interface definition using the `interface` keyword:

```
package example

 interface ILoadable {
    function load()
 }
```

Next, a class can implement the interface with the `implements` keyword followed by a comma-delimited list of interfaces. Implementing an interface means to create a class that contains all methods in the interface:

```
package example

class LoadableThing implements ILoadable {

   function load() {
     print("Loading...")
   }
 }
```

For more information, see "Interfaces" on page 21.

List and Array Expansion Operator *.

Gosu includes a special operator for array expansion and list expansion. This array and list expansion can be useful and powerful. It expands and flattens complex object graphs and extracts one specific property from all objects several levels down in an object hierarchy. The expansion operator is an asterisk followed by a period, for example:

```
names*.Length
```

If you use the expansion operator on a list, it gets a property from every item in the list and returns all instances of that property in a new list. It works similarly with arrays.

Let us consider the previous example `names*.Length`. Assume that `names` contains a list of `String` objects, and each one represents a name. All `String` objects contain a `Length` field. The result of the above expression would be a list containing the same number of items as in the original list. However, each item is the length (the `String.Length` property) of the corresponding name.

Gosu infers the type of the list as appropriate parameterized type using Gosu generics, an advanced type feature. For more information about generics, see "Generics in Gosu" on page 30. Similarly, Gosu infers the type of the result array if you originally call the operator on an array.

This feature also works with both arrays and lists. For detailed code examples, see "List and Array Expansion (*.)" on page 235.

Comparisons

In general, the comparison operators work you might expect if you were familiar with most programming languages. There are some notable differences:

- The operators >, <, >=, and <= operators work with all objects that implement the `Comparable` interface, not just numbers.
- The standard equal comparison == operator implicitly uses the `equals` method on the first (left most) object. This operator does not check for pointer equality. It is `null` safe in the sense that if either side of the operator is null, Gosu does not throw a null pointer exception. (For related information, see “Property Accessor Paths are Null Safe” on page 20.)

  Note: In contrast, in the Java language, the == operator evaluates to `true` if and only if both operands have the same exact **reference value**. In other words, it evaluates to `true` if they refer to the same object in memory. This works well for primitive types like integers. For reference types, this usually is not what you want to compare. Instead, to compare *value equality*, Java code typically uses `object.equals()`, not the == operator.

- There are cases in which you want to use identity reference, not simply comparing the values using the underlying `object.equals()` comparison. In other words, some times you want to know if two objects literally reference the same in-memory object. Gosu provides a special equality operator called === (three equals signs) to compare object equality. It always compares whether both references point to the same in-memory object. The following examples illustrate some differences between == and === operators:

Expression	Prints this Result	Description

`var x = 1 + 2`
`var s = x as String`
`print(s == "3")`
	`true`	These two variables reference the same value but different objects. If you use the double-equals operator, it returns `true`.

`var x = 1 + 2`
`var s = x as String`
`print(s === "3")`
	`false`	These two variables reference the same value but different objects. If you use the triple-equals operator, it returns `false`.

Case Sensitivity

Gosu language itself is case insensitive, but Gosu compiles and runs faster if you write all Gosu as case-sensitive code matching the declaration of the language element. Additionally, proper capitalization makes your Gosu code easier to read. For more information, including Gosu standards for capitalizing your own language elements, see “Gosu Case Sensitivity” on page 31.

Compound Assignment Statements

Gosu supports all operators in the Java language, including bit-oriented operators. Additionally, Gosu has compound operators such as:

- `++` , which is the increment-by-one operator, supported only after the variable name
- `+=` , which is the add-and-assign operator, supported only after the variable name followed by a value to add to the variable
- Similarly, Gosu supports `--` (decrement-by-one) and `-=` (subtract-and-assign)
- Gosu supports additional compound assignment statements that mirror other common operators. See “Variable Assignment” on page 82 for the full list.

For example, to increment the variable `i` by 1:

```
i++
```

It is important to note that these operators always form statements, not expressions. This means that the following Gosu is valid:

```
var i = 1
while(i < 10) {
  i++
  print( i )
}
```

However, the following Gosu is invalid because statements are impermissible in an expression, which Gosu requires in a `while` statement:

```
var i = 1
while(i++ < 10) { // Compilation error!
  print( i )
}
```

Gosu supports the increment and decrement operator only **after** a variable, not before a variable. In other words, `i++` is valid but `++i` is invalid. The `++i` form exists in other languages to support expressions in which the result is an expression that you pass to another statement or expression. As mentioned earlier, in Gosu these operators do not form an expression. Thus you cannot use increment or decrement in `while` declarations, `if` declarations, and `for` declarations.

See “Variable Assignment” on page 82 for more details.

Delegating Interface Implementation with Composition

Gosu supports the language feature called *composition* using the `delegate` and `represents` keywords in variable definitions. Composition allows a class to delegate responsibility for implementing an interface to a different object. This compositional model allows easy implementation of objects that are proxies for other objects, or encapsulating shared code independent of the type inheritance hierarchy. The syntax looks like the following:

```
package test

class MyWindow implements IClipboardPart {
  delegate _clipboardPart represents IClipboardPart

  construct() {
    _clipboardPart = new ClipboardPart( this )
  }
}
```

In this example, the class definition uses the `delegate` keyword to delegate implementation of the `IClipboardPart` interface. The constructor creates a concrete instance of an object (of type `ClipboardPart`) for that class instance variable. That object must have all the methods defined in the `IClipboardPart` interface.

You can use a delegate to represent (handle methods for) **multiple interfaces** for the enclosing class. Instead of providing a single interface name, specify a comma-separated list of interfaces. For example:

```
private delegate _employee represents ISalariedEmployee, IOfficer
```

The Gosu type system handles the type of the variable in the previous example using a special kind of type called a *compound type*.

For more information, see “Composition” on page 195.

Concurrency

If more than one Gosu thread interacts with data structures that another thread needs, you must ensure that you protect data access to avoid data corruption. Because this topic involves concurrent access from multiple threads, this issue is generally called *concurrency*. If you design your code to safely get or set concurrently-accessed data, your code is called *thread safe*.

Gosu provides the following concurrency APIs:

- **Support for Java monitor locks, reentrant locks, and custom reentrant objects.** Gosu provides access to Java-based classes for monitor locks and reentrant locks in the Java package `java.util.concurrent`. Gosu makes it easier to access these classes with easy-to-read `using` clauses that also properly handle cleanup if exceptions occur. Additionally, Gosu makes it easy to create custom Gosu objects that support an easy-to-read syntax for reentrant object handling (see following example). The following Gosu code shows the compact readable syntax for using Java-defined reentrant locks using the `using` keyword. For example:

```
// in your class definition, define a static variable lock
static var _lock = new ReentrantLock()

// a property get function uses the lock and calls another method for the main work
property get SomeProp() : Object
  using( _lock ) {
    return _someVar.someMethod() // do your work here and Gosu synchronizes it, and handles cleanup
  }
```

- **Scoping classes (pre-scoped maps).** Scope-related utilities in the class `gw.api.web.Scopes` help synchronize and protect access to shared data. These APIs return `Map` objects into which you can safely get and put data using different scope semantics.
- **Lazy concurrent variables.** The `LazyVar` class (in `gw.util.concurrent)` implements what some people call a *lazy variable*. This means Gosu constructs it only the first time some code uses it. For example the following code is part of a class definition that defines the object instance. Only at run time at the first usage of it does Gosu run the Gosu block that (in this case) creates an `ArrayList`:

```
var _lazy = LazyVar.make( \-> new ArrayList<String>() )
```

- **Concurrent cache.** The `Cache` class (in `gw.util.concurrent)` declares a cache of values you can look up quickly and in a thread-safe way. It declares a concurrent cache similar to a Least Recently Used (LRU) cache. After you set up a cache object, to use it just call its `get` method and pass the input value (the key). If the value is in the cache, it simply returns it from the cache. If it is not cached, Gosu calls the block and calculates it from the input value (the key) and then caches the result. For example:

```
print(myCache.get("Hello world")
```

For more information about concurrency APIs, see “Concurrency” on page 311

Exceptions

Gosu supports the full feature set for Java exception handling, including `try/catch/finally` blocks. However, unlike Java, no exceptions are checked. Standard Gosu style is to avoid checked exceptions where possible. You can throw any exception you like in Gosu, but if it is not a `RuntimeException`, Gosu wraps the exception in a `RuntimeException`.

Catching Exceptions

The following is a simple `try/catch/finally`:

```
try {
   user.enter(bar)
 } catch( e ){
   print("failed to enter the bar!")
 } finally {
    // cleanup code here...
}
```

Note that the type of `e` is not explicit. Gosu infers the type of the variable `e` to be `Throwable`.

If you need to handle a specific exception, Gosu provides a simplified syntax to make your code readable. It lets you catch only specific checked exceptions in an approach similar to Java’s `try/catch` syntax. Simply declare the exception of the type of exception you wish to catch:

```
catch( e : ThrowableSubclass )
```

For example:

```
try {
  doSomethingThatMayThrowIOException()
}
catch( e : IOException ) {
  // Handle the IOException
}
```

Throwing Exceptions

In Gosu, throw an exception with the `throw` statement, which is the `throw` keyword followed by an object.

The following example creates an explicit RuntimeException exception:

```
if( user.Age < 21 ) {
  throw new RuntimeException("User is not allowed in the bar")
}
```

You can also pass a non-exception object to the throw statement. If you pass a non-exception object, Gosu first coerces it to a String. Next, Gosu wraps the String in a new RuntimeException. As a consequence, you could rewrite the previous throw code example as the concise code:

```
if( user.Age < 21 ) {
  throw "User is not allowed in the bar"
}
```

Annotations

Gosu annotations are a simple syntax to provide metadata about a Gosu class, constructor, method, class variable, or property. This annotation can control the behavior of the class, the documentation for the class.

This code demonstrates adding a @Throws annotation to a method to indicate what exceptions it throws.

```
class MyClass{

@Throws(java.text.ParseException, "If text is invalid format, throws ParseException")
  public function myMethod() {}
}
```

You can define custom annotations, and optionally have your annotations take arguments. (If there are no arguments, you can omit the parentheses).

You can get annotations from types at run time.

For more advanced usage, annotations can define an *interceptor* for a method. This allows you to wrap a method with code before the method runs, after the method runs, or both. This enables certain types of *aspect programming*.

For more information, se “Annotations and Interceptors” on page 199.

Gosu Templates

Gosu supports in-line dynamic templates using a simple syntax. Use these to combine static text with values from variables or other calculations Gosu evaluates at run time. For example, suppose you want to display text with some calculation in the middle of the text:

```
var s = "One plus one equals ${ 1 + 1 }."
```

If you print this variable, Gosu outputs:

```
One plus one equals 2.
```

Template expressions can include variables and dynamic calculations. Gosu substitutes the run time values of the expressions in the template. The following is an example of a method call inside a template:

```
var s2 = "The total is ${ myVariable.calculateMyTotal() }."
```

At compile time, Gosu ensures all template expression are valid and type safe. At run time, Gosu runs the template expression, which must return a String value or a type that can cast to a String.

In addition to in-line Gosu templates, Gosu supports a powerful file-based approach for Gosu templates with optional parameter passing. Any use of the parameters is validated for type-safety, just like any other Gosu code. For example, use a template to generate a customized notification email, and design the template to take parameters. Parameters could include type safe references to the recipient email address, the sender email address, and other objects. Insert the parameters directly into template output, or call methods or get properties from parameters to generate your customized email report.

For more information, see “Gosu Templates” on page 291.

Native XML and XSD Support

Gosu provides native support for XML. XML files describe complex structured data in a text-based format with strict syntax for easy data interchange. For more information on the Extensible Markup Language, refer to the World Wide Web Consortium web page `http://www.w3.org/XML`.

Gosu can parse XML using an existing XML Schema Definition file (an *XSD file*) to produce a statically-typed tree with structured data. Alternatively, Gosu can read or write to any XML document as a structured tree of untyped nodes. In both cases, Gosu code interacts with XML elements as native in-memory Gosu objects assembled into a graph, rather than as text data.

For more information, see "Gosu and XML" on page 245.

Native Web Service Support

Gosu code can import web services (SOAP APIs) from external systems and call these services as a SOAP client (an API consumer). The Gosu language handles all aspects of object serialization, object deserialization, basic authentication, and SOAP fault handling.

The following example uses a hypothetical geocoding service, named `AddressID`.

```
uses soap.AddressID.entity.FindOptions
uses soap.AddressID.entity.FindAddressSpecification
uses soap.AddressID.entity.Address
uses soap.AddressID.entity.FindRange

// insantiate the SOAP endpoint (the SOAP port)
var api = new soap.AddressID.api.FindServiceSoap()

// set the logging output to SHOW YOU the *full* SOAP outgoing call and SOAP response!
api.setLoggingOutputStream( java.lang.System.out)

// In this example, the API provides a class called FindAddressSpecification
// and an API method called FindAddress

// set up a FindAddressSpecification entitiy
var spec = new FindAddressSpecification()
var address = new Address()
address.PostalCode = "94062"
address.CountryRegion = "USA"
address.Subdivision="CA"
spec.InputAddress = address
spec.DataSourceName = ".NA"
spec.Options = new FindOptions()
spec.Options.Range = new FindRange()
spec.Options.Range.Count = 50
spec.Options.ResultMask = {}

// Actually send the request. This actually does the request
var addr = api.FindAddress(spec)
var coordinates = addr.Results.FindResult.FoundLocation[0].LatLong

print("Latitude: " + coordinates.Latitude)
print("Longitude: " + coordinates.Longitude)
```

For more information, see "Calling Web Services from Gosu" on page 73.

Additionally, you can write your own web services in Gosu and request ClaimCenter to publish them. To publish a web service, add the following line before a Gosu class definition:

```
@WebService
```

For example, this simple class publishes a simple web service with one API method:

```
@WebService
class MyServiceAPI {
  function echoInputArgs(p1 : String) : String {
  return "You said " + p1
  }
}
```

You can now call this web service from an external system.

There are some limitations of what kinds of types you can use as argument types and return types. For more information about writing and testing your web service, see “Web Services (SOAP)” on page 25 and “Publishing a Web Service” on page 32.

Gosu Character Set

As Gosu runs within a Java Virtual Machine (JVM), Gosu shares the same 16-bit Unicode character set as Java.

This allows you to represent a character in virtually any human language in Gosu.

More About the Gosu Type System

This topic further describes the Gosu type system and its advantages for programmers. Gosu is a *statically-typed* language (in contrast to a dynamically-typed language). For statically-typed languages, all variables must be assigned a **type** at compile time. Gosu enforces this type constraint at compile time and at run time. If any code violates type constraints at compile time, Gosu flags this as a compile error. At run time, if your code makes violates type constraints (for example, an invalid object type coercion), Gosu throws an exception.

Static typing of variables in Gosu provides a number of benefits:

- Compile Time Error Prevention
- Intelligent Code Completion
- Type Usage Searching

For significantly more information about the Gosu type system, see the topics:

- “Type System” on page 299
- “Basic Type Coercion” on page 299
- “Variable Type Declaration” on page 82.

Although Gosu is a statically-typed language, Gosu supports a concept of generic types, called *Gosu generics*. You can use generics in special cases to define a class or method so that it works with multiple types of objects. Gosu generics are especially useful to design or use APIs that manipulate *collections* of objects. For a summary, see “Generics in Gosu” on page 30, or the full topic “Gosu Generics”, on page 221. Programmers familiar with the Java implementation of generics quickly become comfortable with the Gosu implementation of generics.

Compile Time Error Prevention

Static typing allows you to detect most type-related errors at compile time. This increases reliability of your code at run time. This is critical for real-world production systems. When the Gosu editor detects compilation errors and warnings, it displays them in the user interface as you edit Gosu source code.

For example, functions (including object methods) take parameters and return a value. The information about the type of each parameter and the return type is known at compile time. During compilation, Gosu enforces the following constraints:

- calls to this function must take as parameters the correct number of parameters and the appropriate types.
- within the code for the function, code must always treat the object as the appropriate type. For example, you can call methods or get properties from the object, but only methods or properties declared for that compile-time type. It is possible to cast the value to a different type, however. If the run time type is not a subtype of the compile-time type, it is possible to introduce run time errors.
- for code that calls this function, if it assigns a variable to the result of this function, the variable type must match the return type of this function

For example, consider the following function definition.

```
public function getLabel( person: Person ) : String {
  return person.LastName + ", " + person.FirstName
}
```

For instance, if any code tried to call this method and pass an integer instead of a `Person`, the code fails with a "`type mismatch`" compiler error. That is because the parameter value is not a `Person`, which is the contract between the function definition and the code that calls the function.

Similarly, Gosu ensures that all property access on the `Person` object (`LastName` and `FirstName` properties) are valid properties on the class definition of `Person`. If the code inside the function called any methods on the object, Gosu also ensures that the method name you are calling actually exists on that type.

Within the Gosu editor, any violations of these rules become compilation errors. This means that you can find a large class of problems at compile time rather than experience unpleasant surprises at run time.

Type Inference

As mentioned earlier, Gosu supports type inference, in which Gosu sometimes can infer (determine) the type without requiring explicit type declarations in the Gosu code. For instance, Gosu can determine the type of a variable from its initialized value.

```
var length = 12
var list = new java.util.ArrayList()
```

In the first line, Gosu infers the `length` variable has the type `int`. In the second line, Gosu infers the type of the list variable is of type `ArrayList`. In most cases, it is unnecessary to declare a variable's type if Gosu can determine the type of the initialization value.

Gosu supports explicit declarations of the type of the variable during declaration using the syntax:

```
var c : MyClassName = new MyClassName()
```

However, for typical code, the Gosu coding style is to omit the type and use type inference to declare the variable's type.

Another standard Gosu coding style is to use a coercion on the right side of the expression with an explicit type. For example, suppose you used a class called `Vehicle` and it had a subclass `Car`. If the variable `v` has the compile time type `Vehicle`, the following code coerces the variable to the subtype:

```
var myCar = v as Car
```

Intelligent Code Completion

When you type code into the Gosu editor, the editor uses its type system to help you write code quickly, easily, and preserve the constraints for statically typed variables. When you type the "." (period) character, the editor displays a list of possible properties or subobjects that are allowable.

Similarly, the Gosu editor has a **Complete Code** feature. Choose this tool to display a list of properties or objects that could complete the current code where the cursor is. If you try enter an incorrect type, Gosu displays an error message immediately so you can fix your errors at compile time.

See "Working in Guidewire Studio" on page 123 in the *Configuration Guide* a description of how to access and use the Guidewire Studio **Complete Code** and **SmartHelp** features.

Refactoring

Static typing makes it much easier for development tools to perform smart code refactoring.

Type Usage Searching

Guidewire Studio can search for all occurrences of the usage of an object of a particular type. See the “Using Find-and-Replace” on page 170 in the *Configuration Guide* for how to use Guidewire Studio find and replace functionality.

Property Accessor Paths are Null Safe

One notable difference between Gosu and some other languages is that property accessor paths in Gosu are *null tolerant*, also called *null safe*. This affects only expressions separated by period characters that access a series of *instance variables* or *properties*. In other words, the form:

```
obj.Property1.Property2.Property3
```

In most cases, if any object to the left of the period character is `null`, Gosu does not throw a *null pointer exception* (NPE) and the expression returns `null`. Gosu null-safe property paths tends to simplify real-world code. Often, a `null` expression result has the same meaning whether the final property access is `null` or whether earlier parts of the path are `null`. For such cases in Gosu, do not bother to check for `null` value at every level of the path. This makes your Gosu code easier to read and understand.

For example, suppose you had a variable called `house`, which contained a property called `Walls`, and that object had a property called `Windows`. The syntax to get the `Windows` value is:

```
house.Walls.Windows
```

In some languages, you must worry that if `house` is `null` or `house.Walls` is `null`, your code throws a `null` pointer exception. This causes programmers to use the following common coding pattern:

```
// initialize to null
var x : ArrayList<Windows> = null

// check earlier parts of the path for null to avoid a null pointer exceptions (NPEs)
if( house != null and house.Walls != null ) {
   x = house.Walls.Windows
}
```

In Gosu, if earlier parts of a pure property path are `null`, the expression is valid and returns `null`. In other words, the following Gosu code is equivalent to the previous example and a null pointer exception never occurs:

```
 var x = house.Walls.Windows
```

However, method calls are not null safe. This means that if the right side of a period character is a method call, Gosu throws a null pointer exception if the left side of the period is `null`.

For example, suppose you have a method access followed by a property/variable access:

```
myaction().Walls
```

If the `myaction` method returns `null`, the property access that follows is null safe, and simply returns `null`. Because Gosu assumes property and variable access does not have side effects, it is safe to skip the property access and just return `null`.

In contrast, a **method call** throws an exception if the method’s target (the left side of the period character) is `null`. For example, consider this expression:

```
house.Walls.myaction()
```

If `house` is `null` or `house.Walls` is `null`, Gosu throws an NPE exception. Gosu assumes that method calls might have side effects, so Gosu cannot quietly skip the method call and return `null`.

The Gosu support for null-tolerant property accessor paths are a good reason to expose data as *properties* in Gosu classes and interfaces rather than as setter and getter methods.

Primitives and Null-Safe Paths

Gosu null-safety for property paths does not work if the type of the entire expression is a Gosu primitive type. This is equivalent to saying it does not work if the type of the last item in the series of property accesses has a primitive type. For example, suppose you use the property path:

```
a.P1.P2.P3
```

If the type of the `P3` property is `int` or `char` or other primitive types, then Gosu cannot transparently return `null` if any of the following are `null`:

- `a`
- `a.P1`
- `a.P1.P2`

Primitive types (in contrast to object types, which are descendents of `Object`) can never contain the value `null`. Thus Gosu cannot return `null` from that expression, and any casting from `null` to the primitive type would be meaningless. Therefore, Gosu must throw an NPE exception in those conditions.

For more information about property accessor paths and designing your APIs around this feature, see "Property Paths are Null Tolerant" on page 174.

Generics in Gosu

Generics are a way of abstracting the behavior of a type to support multiple types of objects. Generics are particularly useful for implementing collections (lists, maps) in a type-safe way at compile time by specifying a specific type for its items. For example, instead of just referring to a list of objects, you can refer to a list of `Address` objects or a list of `Vehicle` objects. To use a class with generics, add one or more parameters types inside angle brackets (< and >). For example:

```
uses java.util.*

var mylist = new ArrayList<Date>()
var mymap = new Map<String, Date>()   // a map that maps String to Date
```

Read `ArrayList<Date>` as "a list of date objects".

Read `Map<String, Date>` as "a map that maps `String` to `Date`".

The Gosu generics implementation is 100% compatible with the Java 1.5 generics implementation, and adds additional improvements:

- Gosu type inference greatly improves readability. You can omit unnecessary type declarations, which is especially important for generics because the syntax tends to be verbose.
- Gosu generics support array-style variance of different generic types. In Java, this is a compilation error, even though it is natural to assume it works.

  In Java, this is a compilation error:

  ```
  ArrayList<Object> mylist;
  mylist = new ArrayList<String>()
  ```

  The analogous Gosu code works:

  ```
  var mylist : ArrayList<Object>
  mylist = new ArrayList<String>()
  ```

- Gosu types preserve generic type information at run time. This Gosu feature is called *reified generics*. This means that in complex cases you could check the exact type of an object at run time, including any parameterization. In contrast, Java discards this information completely after compilation, so it is unavailable at run time.

  Note: Even in Gosu, parameterization information is unavailable for all native Java types because Java does not preserve this information beyond compile time. For example the run time type of `java.util.List<Address>` in Gosu returns the unparameterized type `java.util.List`.

- Gosu includes shortcut initialization syntax for common collection types so you do not need to actually see the generics syntax, which tends to be verbose. For example, consider the following Gosu:

```
var strlist = {"a", "list", "of", "Strings"}
```

  Despite appearances, the `strlist` variable is statically typed. Gosu detects the types of objects you are initializing with and determines using type inference that `strlist` has the type `java.util.ArrayList<java.util.String>`. This is generics syntax for the meaning "a list of `String` objects".

For more information, see "Gosu Generics" on page 221.

Gosu Primitives Types

Gosu supports the following primitive types: `int`, `char`, `byte`, `short`, `long`, `float`, `double`, `boolean`, and the special value that means an empty object value: `null`. This is the full set that Java supports.

Additionally, every Gosu primitive type (other than the special value `null`) has an equivalent object type defined in Java. For example, for `int` there is the `java.lang.Integer` type that descends from the `Object` class. This category of object types that represent the equivalent of primitive types are called *boxed primitive* types. In contrast, primitive types are also called *unboxed primitives*. In most cases, Gosu converts between boxed and unboxed primitive as needed for typical use. However, they are slightly different types, just as in Java, and on rare occasion these differences are important. Refer to "Type Object Properties" on page 306 for details.

Custom Type Loaders

At a low level, the Gosu type system supports custom type loaders. Other modules of code can create entire namespaces of new types. This means that a type loader can import external objects and let Gosu code manipulate them as native objects.

There are two custom type loaders that included in ClaimCenter:

- **Gosu XML typeloader.** This type loader supports the native Gosu APIs for XML.
- **Gosu SOAP typeloader.** This type loader supports the native Gosu APIs for SOAP.

Gosu Case Sensitivity

It is best to always use proper (case-sensitive) capitalization for all Gosu code. The Gosu language itself is case insensitive in nearly all cases. However, Gosu compiles and runs faster if you write all Gosu as case-sensitive code matching the declaration of language elements. Additionally, proper capitalization makes your Gosu code easier to read.

The following table lists various language elements and the standard Gosu capitalization for those language elements:

Language element	Standard capitalization	Example
type names	uppercase first character	`DateUtil` `Claim`
variable names	lowercase first character	`myClaim`
method names	lowercase first character	`printReport`
property names	uppercase first character	`Name`

Language element	Standard capitalization	Example
package names (case sensitive)	lowercase entire package name when creating new packages Always specify package names correctly as they are declared. Package names are case sensitive.	`com.mycompany.*`
Java types (case sensitive)	Java types require case sensitivity Always specify Java types correctly as they are declared.	`java.util.String`

Remember to access these items exactly as they are declared.

For example, if an object has a `Name` property, do not write:

```
var n = myObject.name
```

Instead, use the code:

```
var n = myObject.Name
```

Similarly, use class names properly. Do not write:

```
var a = new address()
```

Instead, use the code:

```
var a = new Address()
```

Capitalization in the *middle* of a word is also important. Do not write:

```
var date1 = gw.api.util.DateUtil.currentdate()
```

Instead, use the code

```
var date2 = gw.api.util.DateUtil.currentDate()
```

Guidewire strongly recommends changing any existing code to be case sensitive code, and write any new code to follow these guidelines.

IMPORTANT Guidewire **strongly recommends** you write all Gosu as case-sensitive for all type names, variable names, keywords, method names, property names, package names, and other language elements. If you do not, your code compiles slower, runs slower, and requires more memory at compile time and at run time.

However, Gosu expressions and code executes case-insensitively in this release. That means that effectively there is no difference in behavior in the following two statements in Gosu even though the method `currentDate` has two different case variants:

```
// Valid expressions
var date1 = gw.api.util.DateUtil.currentDate()
var date2 = gw.api.util.DateUtil.currentdate()
```

If you define an item using a case variation of a previously defined item, the Gosu editor displays an error message that the item was previously defined.

```
var date1 = gw.api.util.DateUtil.currentDate()
var date2 = gw.api.util.DateUtil.currentdate()

// this is an invalid expression:
var Date1 = gw.api.util.DateUtil.currentDate() //date1 previously defined!
```

At run time, Gosu treats variables `date1` and `Date1` the same in Gosu code. Therefore the second variable definition (`Date1`) is invalid.

Use the Studio Code Completion feature to enter standard business entities, objects and properties. (See “Working in Guidewire Studio” on page 123 in the *Configuration Guide* for details of code completion.) This ensures a standard for capitalization of business entities, objects, and properties that you can use.

Remember to use initial lower-case for your own variables (local variables and class variables). Use an initial uppercase letter for type names and property names, and initial lowercase letters for method names.

Gosu Statement Terminators

The recommended way to terminate a Gosu statement and to separate statements is:

- a new line character, also known as the invisible `\n` character

Although not recommended, you may also use the following to terminate a Gosu statement:

- a semicolon character (`;`)
- white space, such as space characters or tab characters

In general, use new line characters to separate lines so Gosu code looks cleaner.

For typical code, omit semicolons as they are unnecessary in almost all cases. It is standard Gosu style to use semicolons between multiple Gosu statements when they are all on one line. For example, as in a short Gosu *block* definition (see “Gosu Blocks”, on page 213). However, even in those case semicolons are optional in Gosu.

Valid and Recommended

```
//separated with newline characters
print(x)
print(y)

// if defining blocks, use semicolons to separate statements
var adder = \ x : Number, y : Number -> { print("I added!"); return x + y; }
```

Valid But Not Recommended

```
// NO - do not use semicolon
print(y);

// NO - generally do not rely on whitespace for line breaks. It is hard to read and errors are common.
print(x) print(y)

// NO - generally do not rely on whitespace for line breaks. It is hard to read and errors are common.
var pnum = Policy.PolicyNumber cnum = Claim.ClaimNumber
```

IMPORTANT Generally speaking, omit semicolon characters in Gosu. Semicolons are unnecessary in almost all cases. However, standard Gosu style to use semicolons between multiple Gosu statements on one line (such as in short Gosu block definitions).

Invalid Statements

```
var pnum = Policy.PolicyNumbercnum = Claim.ClaimNumber
```

Gosu Comments

Guidewire strongly recommends that you comment your Gosu code as you create it. The following table lists the comment styles that Gosu supports.

Block	Use block comments to provide descriptions of classes and methods:
	`/*`
	`* The following is a block comment`
	`* This is good for documenting large blocks of text.`
	`*/`

Single-line, with closing markers	Use single-line comments to insert a short comment about a statement or function, either on its own line or embedded in or after other code
	`if(condition) {`
	`  /* Handle the condition. */`
	`  return true    /* special case */`
	`}`

Single-line short comment	Use end-of-line comments (`//`) to add a short comment on its own line or at the end of a line. Add this type of comment marker (`//`) before a line to make it inactive. This is also known as *commenting out* a line of code.
	`var mynum = 1 // short comment`
	`// var mynumother var= 1     // this whole line is commented out -- it does not run`

For more information on using Gosu comments with rules, see “Adding Comments to Rules” on page 35 in the ClaimCenter Rules Guide.

Gosu Reserved Words

Gosu reserves a number of keywords for specialized use. The following list contains all the keywords recognized by Gosu. Gosu does not use all of the keywords in the following table in the current Gosu grammar, and in such cases they remain reserved for future use.

- `application`
- `as`
- `break`
- `case`
- `catch`
- `class`
- `continue`
- `default`
- `do`
- `else`
- `eval`
- `except`
- `execution`
- `extends`
- `finally`
- `final`
- `find`
- `for`
- `foreach`
- `new`
- `null`
- `override`
- `package`
- `private`
- `property`
- `protected`
- `public`
- `readonly`
- `request`
- `return`
- `session`
- `set`
- `static`
- `super`
- `switch`
- `this`
- `try`
- `typeas`

- `function`
- `get`
- `hide`
- `implements`
- `index`
- `interface`
- `internal`
- `native`
- `typeis`
- `typeof`
- `unless`
- `uses`
- `var`
- `void`
- `while`
-

Gosu Generated Documentation

You might want to use the Gosu documentation that you can generate from the command line using the `gwcc` tool:

```
gwcc regen-gosudoc
```

You will find the documentation at `ClaimCenter/build/gosudoc/index.html`.

This documentation is formatted in Javadoc style and contains the output of the Gosu type system. This documentation includes more hyperlinks between objects than using the Gosu API Reference from within Studio.

You can optionally hide certain types from this documentation. To configure this, edit the `gosudoc.properties` file in Studio in the Other Resources section. Your local version (after copy-on-edit to your configuration module) appears at the following path:

```
ClaimCenter/modules/configuration/etc/gosudoc.properties
```

The format of the file is as follows.

Initially, by default, all non-excluded types are entry points using the following line:

```
entrypoint.all=.*
```

After that, you can automatically exclude some types from the output. Any references to these types will be rendered as plain text, with no HTML link. The syntax is:

```
exclude.NAME=PATTERN
```

The *`NAME`* value is a name for this rule. Make it a meaningful name, but it is only for convenience and does not affect the output. The only requirement is that each name must be unique within this section.

The *`PATTERN`* value is a *regular expression* that must match the fully-qualified type name in its entirety.

For example:

```
exclude.internalclasses=com.guidewire.*
```

Running Programs

The initial program that you run **must** be a Gosu program, which is a `.gsp` file. Gosu code in a program can call out to other Gosu classes and other types. For more information about Gosu programs, see "Gosu Program Structure" on page 321.

IMPORTANT You can run Gosu programs (`.gsp` files) directly from the command line. You cannot run a Gosu class file or other types file directly. If you want to call a Gosu class (or other type of file), make a simple Gosu program that uses your other types. In Java, you would define a `main()` method in a class and tell Java which main class to run. It would call out to other classes as needed. In Gosu, your main `.gsp` file can call any code it needs to, including Gosu or Java classes. If you want to reproduce the Java style, your `.gsp` file can contain a single line that call a `main` method on some important Gosu class or Java class.

The types of Gosu files include:

- **Gosu programs** - provide a way to write small scripts or to launch larger Gosu applications
- **Gosu classes** - the core structure for encapsulating data and functions
- **Gosu enhancements** - add properties methods to existing types
- **Gosu interfaces** - a set of methods that other types (implementing classes) must implement
- **Gosu enumerations** - an encapsulated set of unique identifiers associated with unique numbers
- **Gosu templates** - provide a convenient way to build large text objects

Notable Differences Between Gosu and Java

The following table briefly summarizes notable differences between Gosu and Java, with particular attention to changes in converting existing Java code to Gosu. If the rightmost column says *Required*, this is a change that you must make to port existing Java code to Gosu. If it is listed as *Optional*, that item is either an optional feature, a new feature, or Gosu optionally permits the Java syntax for this feature.

Difference	Java	Gosu	Required change?
General Differences			
Gosu language itself is case insensitive, but Gosu compiles and runs faster if you write Gosu as case-sensitive code. Match the declaration of each language element. See “Case Sensitivity” on page 22.	a.B = c.D B and D must exactly match the field declarations.	a.B = c.D Match the code capitalization to match the property declarations. Other capitalizations work, but are not recommended, such as: `a.b = c.d`	Optional
Omit semicolons in most code. Gosu supports the semicolon, but standard coding style is to omit it. (one exception is in block declarations with multiple statements)	`x = 1;`	`x = 1`	Optional.

Print to console with the `print` function. For compatibility with Java code while porting to Gosu, you can optionally call the Java class `java.lang.System`.	`System.out.println("hello");`	`print("hello")`
		
		`uses java.lang.System`
		`System.out.println("hello world")`
			Optional

For Boolean operators, optionally use more natural English versions. The symbolic versions from Java also work in Gosu.	`(a && b) || c`	`(a and b) or c`
		
		`(a && b) || c`
			Optional

Null-safe property accessor paths (see “Property Accessor Paths are Null Safe” on page 20)	`// initialize to null`
	`ArrayList<Windows> x = null`
	
	`// check earlier parts of`
	`// path for null, to avoid`
	`// null pointer exception`
	`if( house != null and house.Walls != null ) {`
	`x = house.Walls.Windows`
	`}`
		`var x = house.Walls.Windows`	Required for cases in which you rely on throwing null pointer exceptions

Functions and Variables			

Difference	Java	Gosu	Required change?
In function declarations: • use the keyword `function` • list the type after the variable, and delimited by a colon. This is true for both parameters and return types.	`public int addNumbers(int x, String y) { ... }`	`public function addNumbers( x : int, y : String) : int { ... }`	Required

In variable declarations, use the `var` keyword. Typically you can rely on Gosu *type inference* and omit explicit type declaration. To explicitly declare the type, list the type after the variable, delimited by a colon. You can also coerce the expression on the right side, which affects type inference	`Auto c = new Auto()`	Type inference
		`var c = new Auto()`
		Explicit:
		`var c : Auto = new Auto()`
		Type inference with coercion:
		`var c = new Auto() as Vehicle`
			Required

To declare variable argument functions, also called `vararg` functions, Gosu does not support the special Java syntax. In other words, Gosu does not support arguments with "`...`" declarations, which indicates variable number of arguments. Instead, design APIs to use arrays or lists. See the following row regarding calling Java functions with this design.	`public String format(Object... args);`	`public format(Object[] args);`
		
		`var c = format({"aa","bb"})`
			Required

To call variable argument functions, pass an array of the declared type. Internally, in Java, these variable arguments are arrays. Gosu array initialization syntax is useful for calling these types of methods.	`public format(String... args);`	`public format(String[] args)`
		
		`var c = format({"aa","bb"})`
			Required

Gosu supports the unary operator assignment statements `++` and `--`. However:
• only use the operator after the variable (such as `i++`)
• these only form statements not expressions.
 	
There are other supported compound assignment statements, such as `+=`, `-=`, and others. see "Variable Assignment" on page 82.
	`if (++i > 2) {`
	`  //`
	`}`
		`i++`
		`if (i > 2) {`
		`  //`
		`}`
			Required

For static members (static methods and static properties), in Gosu you must qualify the type on which the static member appears. Use the period character to access the member. The type does not need to be *fully* qualified, though.	`cos(PI * 0.5)`	`Math.cos(Math.PI * 0.5)` Note that you do **not** need to fully qualify the type as `java.lang.Math`.	Required if you omit type names in your Java code before static members
Type System			

Difference	Java	Gosu	Required change?

For coercions, use the `as` keyword.
Optionally, Gosu supports Java-style coercion syntax for compatability.
	`int x = (int) 2.1`	`// Gosu style`
		`var x = 2.1 as int`
		
		`//Java compatability style`
		`var x = (int) 2.1`
			Optional

Check if an object is a specific type or its subtypes using `typeis`. This is similar to the Java `instanceof` operator.	`myobj instanceof String`	`myobj typeis String`	Required

Gosu automatically downcasts to a more specific type in `if` and `switch` statements. Omit casting to the specific type. See “Automatic Downcasting for ‘typeis’ and ‘typeof’” on page 302.	`Object x = "nice"`
	`Int sl = 0`
	`if( x instanceof String ) {`
	` sl = ((String) x).length`
	`}`
		`var x : Object = "nice"`
		`var sl = 0`
		`if( x typeis String ) {`
		` sl = x.length // downcast`
		`}`
			Optional

To reference the type directly, use `typeof`. However, any direct comparisons to a type do not match on subtypes. Generally, it is best to use `typeis` for this type of comparison rather than `typeof`.	`myobj.class`	`typeof myobj`	Optional

Types defined natively in Gosu as generic types preserve their type information (including parameterization) at run time, generally speaking. This feature is called *reified generics.* In contrast, Java removes this information (this is called *type erasure*).
From Gosu, Java types lack parameterization even if instantiated in Gosu.
However, for native Gosu types, Gosu preserves type parameterization at run time.
	`List<String> mylist = new ArrayList<String>();`
	`system.out.println(typeof mylist)`
	This prints:
	`List`
		`var mylist = new ArrayList<String>()`
		`print(typeof mylist)`
		This prints:
		`List`
		Note that `String` is a Java type.
		However, for native Gosu types as the main type, Gosu preserves the parameterization as run time type information. In the following example, assume `MyClass` is a Gosu class:
		`var mycustom = new MyClass<String>()`
		`print(typeof mycustom)`
		This prints:
		`MyClass<String>`
			Optional for typical use consuming existing Java types.
			If your code checks type information of native Gosu types, remember that Gosu has reified generics.

Gosu generics support array-style variance of different generic types. In Java, this is a compilation error, even though it is natural to assume it works	In Java, this is a compilation error:
	`ArrayList<Object> mylist;`
	`mylist = new ArrayList<String>()`
		The analogous Gosu code works:
		`var mylist : ArrayList<Object>`
		`mylist = new ArrayList<String>()`
			Optional

In Gosu, type names are first-class symbols for the type. Do not get the `class` property from a type name.	`Class sc = String.class`	`var sc = String`	Required
Defining Classes			
Declare that you use specific types or package hierarchies with the keyword `uses` rather than `import`.	`import com.abc.MyClass`	`uses com.abc.MyClass`	Required

Difference	Java	Gosu	Required change?

To declare one or more class constructors, write them like functions called `construct` but omit the keyword `function`. Gosu does not support Java-style constructors.	`class ABC {`
	`  public ABC ( String id ){`
	` }`
	`}`
		`class ABC {`
		`  construct( id : String ) {`
		`  }`
		`}`
			Required

Control Flow			

The `for` loop syntax in Gosu is different for iterating across a list or array. Use the same Gosu syntax for iterating with any iterable object (if it implements `Iterable`). Optionally add "`index indexVar`" before the close parenthesis to create an additional index variable. This index is zero-based. If the object to iterate across is null, the loop is skipped and there is no exception (as there is in Java).	`int[] numbers = {1,2,3};`
	`for (int item : numbers) {`
	`//`
	`}`
		`var numbers : int[] = {1,2,3};`
		`for (item in numbers) {`
		`  //`
		`}`
			Required

The `for` loop syntax in Gosu is different for iterating a loop an integer number of times. The loop variable contains the a zero-based index.
 	
Gosu does not support the for(*initialize;compare;increment*) syntax in Java. However, you can duplicate it using a `while` statement (see example).
	`for(int i=1; i<20; i++){`
	`  //`
	`}`
		`for (item in 20) {`
		`  //`
		`}`
		
		verbose style:
		`var i = 0`
		`while (i < 20) {`
		`  //`
		`  i++`
		`}`
			Required

Annotations			
Gosu requires argument lists to be in the same format as regular function or method argument lists. Gosu does not support the special *named arguments* calling convention from Java.	`@KnownBreak(targetUser = "user", targetBranch = "branch")`	`@KnownBreak("user", "branch")`	Required if your code uses the named argument convention.
Other Gosu-specific features			

Gosu enhancements, which allow you to add additional methods and properties to any type, even Java types. See "Enhancements" on page 209.	*n/a*	`enhancement StrLenEnhancement : java.lang.String {`
		`  public property get PrettyLength() : String {`
		`    return "length : " + this.length()`
		`  }`
		`}`
			Optional

Difference	Java	Gosu	Required change?
Gosu blocks, which are in-line functions that act like objects. They are especially useful with the Gosu collections enhancements See “Gosu Blocks” on page 213. Blocks can also be useful as a shortcut for implementing one-method interfaces (see “Blocks as Shortcuts for Anonymous Classes” on page 219).	*n/a*	`\ x : Number -> x * x`	Optional
Native XML support and XSD support. See “Gosu and XML” on page 245.	*n/a*	`var e = schema.parse(xmlText)`	Optional
Native support for consuming web services with syntax similar to native method calls. See “Calling Web Services from Gosu” on page 73.	*n/a*	`extAPI.myMethod(1, true, "c")`	Optional
Native `String` templates and file-based templates with type-safe parameters. See “Gosu Templates” on page 291.	*n/a*	`var s = "Total = ${ x }."`	Optional

Gosu uses the Java-based collections APIs but improves upon them:
• Simplified initialization syntax that still preserves type safety.
• Simple array-like syntax for getting and setting values in lists, maps, and sets
• Gosu adds new methods and properties to improve functionality of the Java classes. Some enhancements use Gosu blocks for concise flexible code.
For new code, use the Gosu style initialization and APIs. However, you can call the more verbose Java style for compatibility. See “Collections” on page 231.
		`// easy initialization`
		`var strs = {"a", "ab", "abc"}`
		
		`// array-like "set" and "get"`
		`strs[0] = "b"`
		`var val = strs[1]`
		
		`// new APIs on Java`
		`// collections types`
		`strList.each(\ str -> { print(str) } )`
			Optional

List and array expansion operator. See “List and Array Expansion (*.)” on page 235.	*n/a*	`names*.Length`	Optional

Get Ready for Gosu

As you have read, Gosu is a powerful and easy-to-use object-oriented language. Gosu combines the best features of Java (including compatibility with existing Java libraries), and adds significant improvements like blocks and powerful type inference that change the way you write code. Now you can write easy-to-read, powerful, and type safe type code built on top of the Java platform. To integrate with external systems, you can use native web service and XML support built directly into the language. You can work with XSD types or external APIs like native objects.

For these reasons and more, large companies all around the world use Gosu every day in their production servers for their most business-critical systems.

The next step for you is to write your first Gosu program and become familiar with the Guidewire Studio application and the Gosu editor in Studio. See “Working in Guidewire Studio” on page 123 in the *Configuration Guide* for more information about the Gosu editor in Studio.

chapter 2

Gosu Types

This topic describes the Gosu-supported data types and how to use each one. For more information about manipulating types or examining type information at run time, see “Type System” on page 299.

This topic includes:

Built-in Types

Guidewire Gosu supports the following native data types:

- `Array`
- `Boolean`
- `DateTime`
- `Type`
- `Key`
- `Number`
- `Object`
- `String`

Array

An *array* is a collection of data values, with each element of the array associated with a number or *index*. Example values for the `Array` data type are:

- `Array[]`
- `Number[]`

- `Boolean[]`
- `DateTime[]`
- `Object[]`
- `String[]`

The following example shows a standard indexed array:

```
myValue = Claim.Exposures[1]
```

Alternatively, you can use an *associative array*, which uses a `String` value as the index instead of a number. Instead of accessing an array element directly through its index number, you access it with a string value that has been set to an index value. Associative arrays are useful if an index value is not known at compile time, but can be determined at run time, as with user-entered data, for example.

```
myValue = Claim["ClaimNumber"]
```

Associative arrays have some similarities to the Java language `Map`, which is also supported in Gosu.

For more information creating and using arrays, see “Arrays” on page 52.

For more information about special Gosu APIs related to using lists, maps, and other collections in Gosu, see “Collections”, on page 231.

Boolean

Gosu contains two types that can contain the values `true` and `false`:

- A primitive type called `boolean` that corresponds to Java’s primitive `boolean` type. Possible values for variables declared to the `Boolean` data type are:
  - `true`
  - `false`
- A Gosu `Boolean` type that is a *boxed type*, which means it is a class wrapper around a primitive `boolean` value. Possible values for variables declared to the `Boolean` data type are:
  - `true`
  - `false`
  - `null`

For both `boolean` and `Boolean`, some other values can coerce to `true` or `false`. For example, the following values coerce to `true`:

- the number 1
- the `String` value with data `"true"` (see the following discussion)

The following values coerce to `false`:

- the number 0
- the `String` value with data `"false"` or any other value other than `"true"` (see the following discussion).

It is important to note the value `null` is not the same as `false` and coercions for this value work differently between the two types. Variables and properties with the `Boolean` type can have a value of `null`, where `null` means “I do not know” or “unstated” in addition to `true` and `false`. However, variables and properties with the `boolean` type cannot have a value of `null`, and can only coerce to `true` or `false`.

Because of this, there are important differences in coercing object types:

- `null` coerced to a variable of type `Boolean` stores the original value `null`.
- `null` coerced to a variable of type `boolean` stores the value `false` because primitive types cannot be `null`.

If you coerce `String` data to either `Boolean` or `boolean`, the `String` data itself is examined. It coerces to `true` if and only if the text data has the value `"true"`. Be careful to check for `null` values for `String` variables as appropriate to avoid ambiguity in how your code handles `null` values. A `null` value may indicate uninitialized data or other unexpected code paths.

IMPORTANT For important information about primitives and comparisons of boxed and unboxed types in Gosu, see "Working with Primitive Types" on page 306.

Example

```
var hasMoreMembers == null
var isDone = false
```

DateTime

`DateTime` data types involve, as the name suggests, values that are either calendar dates or time (clock) values, or both. The following table lists possible formats for the `DateTime` data type.

Format	Example
`MMM d, yyyy`	`Jun 3, 2005`
`MM/dd/yy`	`10/30/06`
`MM/dd/yyyy`	`10/30/2006`
`MM/dd/yy hh:mm a`	`10/30/06 10:20 pm`
`yyyy-MM-dd HH:mm:ss.SSS`	`2005-06-09 15:25:56.845`
`yyyy-MM-dd HH:mm:ss`	`2005-06-09 15:25:56`
`yyyy-MM-dd'T'HH:mm:ssz`	`2005-06-09T15:25:56 -0700`
`EEE MMM dd HH:mm:ss zzz yyyy`	`Thu Jun 09 15:24:40 GMT 2005`

Individual characters in the previous table have the following meaning:

Character	Meaning
`a`	AM or PM (determined from 24-hour clock)
`d`	day
`E`	Day in week (abbreviation)
`h`	hour (24 hour clock)
`m`	minute
`M`	month
`s`	second
`S`	fraction of a second
`T`	parse as time (ISO8601)
`y`	year
`z`	Time Zone offset (GMT, PDT, and so on)

Other possible values are:

- `null`
- 1124474955498 (milliseconds since 12:00:00:00 a.m. 1/1/1970 UTC)

If you do not specify the time (as in `"October 31, 2002"`), then Gosu sets the implied time to 12:00 a.m. of that day. Use the built-in Gosu `gw.api.util.DateUtil.*` library functions to work with `DateTime` objects. There are methods to add a certain number of days, weekdays, weeks, or months to a date. There is a method to remove the time element from a date (`trimToMidnight`). Type `gw.api.util.DateUtil` into the Gosu Tester, and then press period to see the full list of methods.

See the *ClaimCenter Rules Guide* for information about the Guidewire-provided Gosu library functions.

Example

```
var diff = gw.api.util.DateUtil.daysBetween( "Mar 5, 2006", gw.api.util.DateUtil.currentDate() )
```

Gosu DateTime and Java

Gosu represents `DateTime` objects internally using `java.util.Date`. However, this is internal only and thus it is not possible to access the (Java) `Date` type directly. Instead, Gosu's `DateTime` type exposes the functionality of the `Date` methods using operators. For example:

- Gosu supports the `Date` `before()` and `after()` methods as relational operators. In Gosu, use the following to test whether one date is after another date:
  ```
  date1 > date2
  ```
- Gosu supports the `Date` `getTime()` method using the cast operator. In Gosu, use the following to determine the number of milliseconds between a `DateTime` object and January 1, 1970, 00:00:00 GMT:
  ```
  date1 as Number
  ```

Gosu implicitly coerces the Gosu `DateTime` object from `String` in most formats. For example:

```
var date : DateTime = "2007-01-02"
```

Key

`Key` is a Guidewire data model entity base type that uniquely identifies an entity. For example, entity IDs such as the internal ID of a `Claim` are of type `Key`. Sometimes functions or internal domain methods require `Key` objects as parameters, so it is often useful to cast an entity to a key.

Example

```
var id = Claim as Key
```

Note: Do not confuse the very different types `Key` and `Typekey`. A typekey refers to one element in a typelist for multiple choice values within an entity. See "Typekeys" on page 57.

Number

The `Number` data type represents all numbers, including integers and floating-point values. Gosu supports the standard Java number types of `double`, `float`, `int`, `long`, and `short`. (See "Primitive Types" on page 51.) Possible values for the `Number` data type are:

- `1`
- `246`
- `3.14159`
- `NaN`
- `Infinity`
- `null`
- `"9.2"`

Gosu converts a `String` value that contains only numbers to a number.

IMPORTANT For more information about primitives and comparisons of boxed and unboxed types in Gosu, see "Working with Primitive Types" on page 306.

Scientific Notation

Guidewire Gosu supports the use of scientific notation to represent large or small numbers. Scientific notation represents a number as a *coefficient* (a number greater than or equal to 1 and less than 10) and a *base* (which is always 10). For example, consider the following number:

1.23×10^{11}

The number 1.23 is the coefficient. The number 11 is the exponent, which means the power of ten. The base number 10 is always written in exponent form. Gosu represents the base number as the letter “e”, which stands for “exponent”.

You must enclose a number written in scientific notation within quotation marks, such as "2.057e3", for Gosu to understand the number.

Examples

```
var PI= 3.14
var count = 0
var result1 = "9.2" * 3
var result2 = "2.057e3" * PI

//Result
result1 = 27.599999999999998
result2 = 6458.9800000000005
```

Object

An object encapsulates some data (variables and properties) and methods (functions).

Internally, an object has an intrinsic type that encapsulates the underlying data source. Gosu connects directly with a variety of data sources through intrinsic types. Java classes are represented internally through a Java intrinsic type that bridges Gosu with Java classes.

Similarly, data model entities are exposed through an `Entity` intrinsic type that reflects the Guidewire product data model configuration. Many other intrinsic types exist to handle various other kinds of data sources. All the intrinsic types are unified through the `Object` type.

Examples

```
var a : Address
var map = new java.util.HashMap()
```

For more information creating and using objects, see “Object Instantiation and Properties” on page 54.

For information about different aspects of the business object data model that define business entities, see “The ClaimCenter Data Model” on page 187 in the *Configuration Guide*.

For more information about the Gosu type system, see “Type System” on page 299.

String

Gosu treats strings as a sequence of characters. You create a string by enclosing a string of characters in beginning and ending double-quotation marks. Possible values for the `String` data type are:

- `"homeowners"`
- `"auto"`
- `""`

String Variables Can Have Content, Be Empty, or Be Null

It is important to understand that the value `null` represents the absence of an object reference and it is distinct from the empty `String` value `""`. The two are not interchangeable values. It is important to remember that a variable declared to type `String` can hold the value `null`, the empty `String` (`""`), or a non-empty `String`.

There is a simple way in Gosu to test for a populated `String` object versus a `null` or empty `String` object. Use the `HasContent` method. When you combine it with the null-tolerant property access in Gosu, it will return false if the value is `null` or an empty `String` object. Compare the behavior of properties `HasContent` and `Empty`:

```
var s1:String=null
var s2:String=""
var s3:String="hello"

print("has content = " + s1.HasContent)
print("has content = " + s2.HasContent)
print("has content = " + s3.HasContent)

print("is empty = " + s1.Empty)
print("is empty = " + s2.Empty)
print("is empty = " + s3.Empty)
```

This code outputs:

```
has content = false
has content = false
has content = true
is empty = false
is empty = true
is empty = false
```

Whether the variable holds an empty `String` or `null`, the method returns `false` as expected.

String Properties in Entity Instances Have Special Setting Behavior

When you **set** a property on an entity instance, there is a special behavior that is different from properties on other kinds of objects like Gosu and Java classes.

When you set a `String` value to an entity property that has the Gosu type `String`, first Gosu removes spaces from the beginning and end of the `String` value. Next, if the result is the empty `String` (""), Gosu sets the value `null` instead of the empty `String`.

For example:

```
var obj = new temp1()   // new normal Gosu or Java object
var entityObj = new Address()  // new entity instance

// set String property on a regular object
obj.City = "           San Francisco      "
display("Gosu", obj.City)
obj.City = "    "
display("Gosu", obj.City)
obj.City = null
display("Gosu", obj.City)

// set String property on an entity instance
entityObj.City = "           San Francisco      "
display("entity", entityObj.City)
entityObj.City = "    "
display("entity", entityObj.City)
entityObj.City = null
display("entity", entityObj.City)


function display (intro : String, t : String) {
print (intro + " object " + (t == null ? "NULL" : "\"" + t + "\""))
}
```

This outputs:

```
Gosu object "           San Francisco      "
Gosu object ""
Gosu object NULL
entity object "San Francisco"
entity object NULL
entity object NULL
```

Note that the entity property has no initial or trailing spaces in the first case, and is set to `null` in the other cases.

If you want to test a `String` variable to see if it has content or is either `null` or empty, use the `HasContent` method. See “String Variables Can Have Content, Be Empty, or Be Null” on page 47.

> **IMPORTANT** When you **set** a `String` property on an entity instance, Gosu automatically trims initial and trailing space characters before setting the property. Also, when the result is an empty `String`, Gosu sets the property to `null` rather than the empty `String`.

Configuring Whitespace Removal for Entity Text Properties

You can control whether ClaimCenter trims whitespace before committing a `String` property to the database with the `trimwhitespace` column parameter in the data model definition of the `String` column. Columns that you define as `type="varchar"` trim leading and trailing spaces by default.

To prevent ClaimCenter from trimming whitespace before committing a `String` property to the database, include the `trimwhitespace` column parameter in its column definition, and set the parameter to `false`.

```
<column
    desc="Primary email address associated with the contact."
    name="EmailAddress1"
    type="varchar">
    <columnParam name="size" value="60"/>
    <columnParam name="trimwhitespace" value="false"/>
</column>
```

Note: The parameter controls only whether ClaimCenter trims leading and trailing spaces. You cannot configure whether ClaimCenter coerces an empty string to `null`.

Other Methods on String Objects

Gosu provides various methods to manipulate strings and characters. For example:

```
var str = "bat"
str = str.replace( "b", "c" )
print(str)
```

This prints:

```
cat
```

String Utilities

You can access additional `String` methods in API library `gw.api.util.StringUtil`. Type `gw.api.util.StringUtil` into the Gosu Tester and press period to see the full list of methods.

In-line String Templates

If you define a `String` literal directly in your Gosu code, you can embed Gosu code directly in the `String` data. This feature is called templates. For example, the following String assignment uses template features:

```
var s = "One plus one equals ${ 1 + 1 }."
```

If you print this variable, Gosu outputs:

```
One plus one equals 2.
```

For more information, see “Gosu Templates”, on page 291.

Escaping Special Characters in Strings

In Gosu strings, the backslash character (\) indicates that the character directly after it requires special handling. As it is used to “escape” the usual meaning of the character in the string, the backslash is called an escape character. The combination of the backslash and its following character is called an escape sequence.

For example, you use the backslash escape character to insert a double-quotation mark into a string without terminating it. The following list describes some common uses for the backslash in Gosu strings.

Sequence	Result
`\\`	Inserts a backslash into the string without forcing an escape sequence.
`\"`	Inserts a double-quotation mark into the string without terminating it. **Note:** This does not work inside embedded code within Gosu templates. In such cases, do not escape the double quote characters. See “Gosu Templates” on page 291.
`\n`	Inserts a new line into the string so that the remainder of the text begins on a new line if printed.
`\t`	Inserts a tab into the string to add horizontal space in the line if printed.

Examples

```
Claim["ClaimNumber"]
var address = "123 Main Street"
"LOGGING: \n\"Global Activity Assignment Rules\""
```

Gosu String Templates

In addition to simple text values surrounded by quote signs, you can embed small amounts of Gosu code directly in the `String` as you define a `String` literal. Gosu provides two different template styles. At compile time, Gosu uses its native type checking to ensure the embedded expression is valid and type safe. If the expression does not return a value of type `String`, Gosu attempts to coerce the result to the type `String`.

Use the following syntax to embed a Gosu expression in the `String` text:

```
${ EXPRESSION }
```

For example, suppose you need to display text with some calculation in the middle of the text:

```
var mycalc = 1 + 1
var s = "One plus one equals " + mycalc + "."
```

Instead of this multiple-line code, embed the calculation directly in the `String` as a template:

```
var s = "One plus one equals ${ 1 + 1 }."
```

If you print this variable, Gosu outputs:

```
One plus one equals 2.
```

This style is the preferred `String` template style.

However, Gosu provides an alternative template style. Use the three-character text `<%=` to begin the expression. Use the two-character text `%>` to end the expression. For example, you can rewrite the previous example as the following concise code:

```
var s = "One plus one equals <%= 1 + 1 %>."

print("one")
var s = "Hello. <% print("two") %>We will go to France<% print("three") %>."
print(s)
```

Within a code expression, do not attempt to escape double quote characters inside templates using the special syntax for quote characters in `String` values. In other words, the following is valid Gosu code:

```
var s= "<% var myvalue = {"a", "b"} %>"
```

However, the following is invalid due to improper escaping of the internal double quotes:

```
var foo = "<% var bar = {\"a\", \"b\"} %>"
```

For much more information about Gosu templates, see “Gosu Templates” on page 291.

Type

The `Type` data type is a meta-type. It is the "type" of types. If you get the type of something using the `typeof` keyword, the type of the result is `Type`.

Examples of types

```
Array
DateTime
Number
String
Type
int
java.util.List[]
```

For more information about using `Type` objects to get information about a type, see "Type System" on page 299.

Note the following aspects of types:

- **Everything has a type.** All Gosu values have a type.
- **Language primitives have types.** For example the code "`typeof 29`" is valid Gosu, and it returns `java.lang.Integer`, which is a `Type`. In other words, `Integer` is a subtype of `Type`.
- **Object instances have types.** The type of an instance of a class is the class itself.
- **Even types have types.** Because everything has a type, you can use `typeof` with `Type` objects also.

> **IMPORTANT** For more information about the `Type` class and the `typeof` keyword how to use it, see "Basic Type Checking" on page 300.

Primitive Types

Guidewire Gosu supports the following Primitive types for compatibility with Java language primitives.

Primitive	Type	Value
`boolean`	Boolean value	`true` or `false` (also possibly `null`)
`byte`	Byte-length integer	8-bit two's complement
`char`	Single character	16-bit Unicode
`double`	Double-precision floating point number	64-bit (IEEE 754)
`float`	Single-precision floating point number	32-bit (IEEE 754)
`int`	Integer	32-bit two's complement
`long`	Long integer	64-bit two's complement
`short`	Short integer	16-bit two's complement

IEEE 754 is the IEEE (Institute of Electrical and Electronics Engineers) standard for Binary Floating-Point Arithmetic. For more information, see the following: `http://en.wikipedia.org/wiki/IEEE_754`.

From Gosu you can also use boxed versions (non-primitive versions) of these primitive types defined for the Java language, in the `java.lang` package.

> **IMPORTANT** For important information about primitives and comparisons of boxed and unboxed types in Gosu, see "Working with Primitive Types" on page 306.

Access to Java Types

Gosu is built on top of the Java language. Because Gosu provides an dynamic type system, Gosu includes a built-in typeloader for all Java types. This means that from Gosu you have full direct access to Java types, such as Java classes and Java libraries. Simply access the type directly from Gosu and it works just like in Java. The access to Java types includes the following:

- instantiate Java objects with the standard Gosu `new` keyword
- call static methods on Java classes
- call object methods on instantiated objects
- get properties from Java objects

Arrays

As described previously ("Gosu Types" on page 43), an array can be either indexed or associative.

- *Indexed* arrays use an index number to access an array member.
- *Associative* arrays are like maps of strings to values. They are only arrays in the sense that you can access elements of the map using array notation dynamically.

If you create an array, you must explicitly define the size of the array or implicitly define the size by simultaneously defining the array elements. For example:

```
// arrays of strings
var s1 = new String[4]
var s2 = new String[ ] {"This", "is", "a", "test."}

// arrays of integers
var a1 = new int[3]
var a2 = new int[] {1,2,3}
var a3 : int[] = {1,2,3}
```

To access the elements of an array, use the following syntax.

Syntax

```
<expression>[<index value>]
<expression>[<associative value>]
```

Examples

Expression	Result
`Claim.Exposures[0]`	An exposure
`gw.api.util.StringUtil.splitWhitespace( "a b c d e" )[2]`	`"c"`

You can also iterate through the members of an array using the `for()` construction. See "Iteration in For() Statements" on page 87 for details.

You can also create a new array with a default value for each item using an included Gosu enhancement on the Arrays object. Call the `makeArray` method and pass it a default value and the size of the array. Gosu uses the type of the object to type the array.

For example, create an array of 10 items initialized to the Boolean value `false` with the code:

```
var x = Arrays.makeArray( false, 10 )
```

Be aware that in situations it may be more appropriate to use collections such as `List` or `Map` than to use arrays. Collections such as `List` or `Map` are inherited from the Java language but have additional enhancement methods in Gosu.

> **IMPORTANT** Consider using collections instead of arrays to take advantage of some advanced features of Gosu. For more information about special Gosu APIs for lists, maps, and other collections in Gosu, see "Collections", on page 231.

Array-related Entity Methods

Guidewire products contain a number of high-level business entities such as a `Claim`, `Exposure`, `Policy`, or `Account`. An entity serves as the root object for data views, rules, and most other data-related areas of the product. XML elements define these data entities. The root element of an entity definition specifies the kind of entity, as well as any the entity's attributes. Subelements of the entity define the entity components, such as columns (properties) and foreign keys. (See the *ClaimCenter Configuration Guide* for information on data entities and arrays.)

You can create an array of other entities associated with the root entity by adding the correct XML in the appropriate `dm_*.xml` (data model) file. For example, in Guidewire ClaimCenter, array definitions in `dm_cc_claim.xml` associate the following arrays with a `Claim` object:

```
<array name="Access" ... />
<array name="Activities" ... />
...
```

Gosu automatically adds a number of `addTo...()` and `removeFrom...()` methods for manipulating array elements if you create an object, based on the arrays defined in XML. (This is true also of arrays added either as extensions of existing entities or arrays that are created on new custom entities, not just on Guidewire-defined entities and arrays.)

For example, Gosu adds the following array methods automatically to a `Claim` object with the arrays defined earlier:

```
addToAccess()
addToActivities()
...
removeFromAccess()
removeFromActivities()
...
```

addTo...() Methods

The `addTo...()` methods set a foreign key that points back to the array owner. You must explicitly call the `addTo...()` method on newly-created array elements. Otherwise, the Rule Engine does not commit the array elements to the database. Guidewire strongly recommends as a "best practice" that you always call the appropriate `addTo..()` method to associate an array element with its owner.

removeFrom...() Methods

The `removeFrom...()` methods behave differently depending on whether the data element is retireable or not:

- If the element is retirable, the `removeFrom...()` methods retire the element.
- If the element is not retireable, the `removeFrom...()` methods delete the element.

The following example removes all elements in array `CCXContribFactors` (claim contributing factors) in which the primary contributing factor was driver error.

```
for (ccxContribFactor in claim.CCXContribFactors) {
  if (ccxContribFactor.contribprimary == "Driver Factors") {
    claim.removeFromCCXContribFactors(ccxContribFactor)
  }
}
```

Additional Methods

Gosu provides a number of additional methods for working with arrays. These include the following:

- `get...()` methods to retrieve specific array elements
- `isArrayElement...()` methods to determine the status of specific array elements

Consult the Gosu API JavaDoc for specific information about these methods.

Java-based Lists as Arrays

Gosu provides a means to access Java language list members using standard array notation. It is necessary to populate the list using one of the `add()` methods before you can use array notation to access it. (See also “For() Statements” on page 87 for more examples of lists and array access.)

Example

```
var list = new java.util.ArrayList()

//Populate the list with values.
list.add("zero")
list.add("one")
list.add("two")
list.add("three")
list.add("four")

//Assign a value to a member.
list[3] = "threeUPDATED"

//Automatically iterate through list members and print.
for ( member in list ) {
  print(member)
}

//Iterate through list members using array notation and print.
for (member in list index i) {
  print(list[i])
}
```

The output for this code is:

```
zero
one
two
threeUPDATED
four
```

In some situations, it may be more appropriate to use *collections* such as `List` or `Map` rather than to use arrays. The `List` and `Map` classes inherit from the Java language and Gosu adds additional enhancement methods. For more information about special Gosu APIs related to using lists, maps, and other collections in Gosu, see “Collections”, on page 231.

Array Expansion

Gosu supports a special operator called *. that expands arrays and lists. For more information, see “List and Array Expansion (*.)” on page 235.

Object Instantiation and Properties

A Gosu object is an instance of a type. A type can be a class or other construct exposed to Gosu through the type system. (A class is a collection of properties and methods.)

Note: See “Built-in Types” on page 43 for a list of valid Gosu types.

Entity instances are special objects defined through the data model configuration files. Gosu stores instances of entities in the database if some code commits the bundle (sends data changes to the database).

Creating New Objects

You use the Gosu new operator to create an instance of a type. For example:

```
new java.util.ArrayList() // Create an instance of an ArrayList.
new Number[5]             // Create an array of numbers.
new LossCause[3]          // Create an array of loss causes.
```

See “New Object Expressions” on page 70 for more details.

Assigning Object Properties

Property assignment is similar to variable assignment.

Syntax

```
<object-property-path> = <expression>
```

However, properties can be write-protected (as well as read-protected). For example, the following Gosu code:

```
Activity.UpdateTime = "Mar 17, 2006"
```

causes the following error:

```
Property, UpdateTime, of class Activity, is not writable
```

Example

```
myObject.Prop = "Test Value"
var startTime = myObject.UpdateTime
```

Accessing Object Properties

Gosu retrieves a property’s value using the dot-separated path rooted at an expression. If any property in the path evaluates to `null`, the entire path evaluates to `null` (instead of throwing an exception and terminating execution).

Syntax

```
<expression>.<property>
```

Examples

Expression	Result
`Claim.Contacts.Attorney.Name`	`Some Name`
`Claim.Addresses.State`	`New Mexico`

Accessing Object Methods

An object property can be any valid data type, including an array, a function, or another object. An object function is generally called a *method*. Invoking a method on an object is similar to accessing an object property, with the addition of parenthesis at the end to denote the function. Gosu uses the dot notation to call a method on a object instance. See the information in “Handling Null Values” on page 78 for a discussion on how Gosu handles object method calls that evaluate to `null`.

Syntax

```
<expression>.<method-call>
```

Example

Expression	Result
`Claim.isClosed()`	`Return a Boolean value indicating the status of Claim`
`Claim.SetFlag()`	`Flag this claim for the specified reason`
`Claim.CreateActivityFromPattern()`	`Associate an activity of the specified type with this claim`

See “Static Method Calls” on page 79 for more details. See also “Using Reflection” on page 304 for regarding using type information to determine methods of an object.

Accessing Object Arrays

Gosu provides access to the properties of an Object by property name or by associative array access. Use the following syntax to access Object properties through array access.

Syntax

```
<object-reference>[<property name>]
```

Array access uses late binding to avoid potential compile-time errors. For example, suppose that you have an instance of a general type, say Object, as a function argument:

```
function getDisplayName( obj : Object) {
  var name : String
  name = obj.DisplayName     // Compile error - no DisplayName property on Object
  name = obj["DisplayName"]  // Ok, so long as there is a DisplayName property at runtime
}
```

The only requirement is that object's type have a property called `DisplayName`; there is no base interface that all the arguments implement. The only way to access the `DisplayName` property without a compile-time error is through “late binding” as with associative array access.

In this example, if the function caller passes an object parameter that does not have a `DisplayName` property, at run time Gosu throws an exception.

Examples

Expression	Result	Description
`person["StreetAddress"]`	`"123 Main Street"`	example of a single associative array access
`person["Address"]["City"]`	`"Birmingham"`	example of a double associative array access. This is equivalent to the code `person.Address.City`.

In addition to being able to read an associative array member, you can write to an associative array member as well. For example:

```
var address : Address
var city = address["City"]
address["City"] = "San Mateo"
```

Entities

Entities are a type of object that is constructed from the data model configuration files. Like other types of objects, entities have data in the form of object properties, and actions in the form of object domain methods.

For information about different aspects of the business object data model, see “The ClaimCenter Data Model” on page 187 in the *Configuration Guide*.

Some Guidewire API methods take an argument of type `IEntityType`. The `IEntityType` type is an interface (not a class) and all Guidewire entity types implement this interface. In other words, an instance of an entity does not implement this interface, but its type implements it. To use an API that requires an `IEntityType`, pass the entity you want to pass to it.

For example, to pass an `Address` to an API that takes an `IEntityType`, use code such as:

```
myClass.myMethod( Address )
```

In rare cases you might need to dynamically get the type from entity, for example if you do not know what subtype the entity is. If you have a reference to an instance of an entity, get the `Type` property from the entity. For example, if you have a variable `a` that contains an `Address` entity, use code such as the following:

```
myClass.myMethod( a.Type )
```

Typekeys

typelists, which are similar to enumerations but have unique codes caled typecodes. For example, the typelist called `AddressType` contains three typekey values, which have the codes `BUSINESS`, `HOME`, and `OTHER`. For information about the ClaimCenter data model, see "The ClaimCenter Data Model" on page 187 in the *Configuration Guide*.

Typelist Literals

In most cases, to get a literal for a typelist simply type the type list name.

In rare cases, there might be ambiguity about the name or package of the typelist in some programming contexts. If this is the case, use the fully qualified syntax `typekey.`*`TYPELISTNAME`*. For example, `typekey.AddressType`.

Typekey Literals

To reference an existing typekey from Gosu, access the typecode with typelist name and the typecode value in capital letters with the `TC_` prefix. In other words, use the syntax:

```
TYPELISTNAME.TC_TYPECODENAME
```

For example, to select from the `AddressType` typelist the typecode with code `BUSINESS`, use the syntax:

```
AddressType.TC_BUSINESS
```

IMPORTANT If you access a typecode using the `TC_` prefix followed by the typekey's code, the proper capitalization is always fully capitalized. This is because the typecode reference is a static instance created at compile time. The capitalization standard for static instance variables is fully capitalized.

Typecode Value Coercions

To get a typekey from its code, use the static instance syntax described earlier in this section using the typelist type and its static instance: *`TYPELISTNAME`*`.TC_`*`TYPECODENAME`*. For example, `State.TC_CA`. That is the preferred approach.

```
myClaim.LossType = "AUTO"
```

If and only if the programming context accepts a specific typecode type, this is a typesafe technique despite appearances. At compile time, Gosu checks that any text you type directly in your code represents a legal typecode at compile time. In other words, if `ZZ` is an invalid typecode for this typelist and you typed `"ZZ"` for that value, Gosu reports at compile time that it is a syntax error .

WARNING Passing a `String` version of the code of a typekey to represent the typekey is not always typesafe. If a programming context can accept a value other than a typekey from a specific typecode value as a `String`, this is not typesafe. Because the context is not always clear, it is preferable to use the *`TYPELISTNAME`*`.TC_`*`TYPECODENAME`* syntax to avoid errors.

The same typesafe coercion happens if you pass arguments to a function and it requires a specific value from one typelist. For example, this approach works if a method accepts only a typecode value `typekey.State`:

```
myinstance.actOnState("CA")
```

Run Time (Non-Typesafe) Coercions

Using the `String` value for the code to refer to a typekey is unsafe, however, if the programming context can accept values other than a specific typecode. For example, if the variable or function argument accepts the object base class `Object`, it is **not typesafe** to pass a `String` of the code for the typekey. At compile time, the Gosu compiler treats it as a normal `String` value because it does not know what typelist to validate the code against. In such cases, instead use the verbose syntax *`TYPELISTNAME.TC_TYPECODENAME.`* For example, Gosu entity query APIs take `Object` references for comparing values to typekeys and thus for type safety requires references to typekeys in the verbose syntax.

If your code must have dynamic `String` data coerced at run time to typekeys, you can explicitly coerce the value:

```
var myTypekey = codeString as typekey.State
```

WARNING If the `String` value is not known at compile time, this explicit coercion is not typesafe. Only use this technique if you must perform run-time conversion of `String` data. If the coercion fails at run time, Gosu throws an exception.

Get All Available Typekeys from a Typelist

You can programmatically get a list of typekeys in a typelist. You can choose whether to get all typekeys, including *retired* typekeys as well as *non-retired* typekeys, or just non-retired typekeys. You can retire a typekey in the data model configuration files.

To get typekeys from a typelist, call the `getTypeKeys` method:

```
TYPENAME.getTypeKeys(includeRetiredTypekeys)
```

The Boolean argument indicates whether to include retired typekeys in the returned list. If set to `true`, retired typekeys are included. Otherwise, return only unretired typekeys.

For example:

```
// prints all typekeys in the AddressType typelist
print(AddressType.getTypeKeys(false))

// prints the code of the first typekey in the array
print(AddressType.TypeKeys[0].Code)
```

This code prints:

```
[Business, Home, Other]
business
```

Getting Information from a Typekey

From Gosu, if you have a reference to a typekey, you can get the following properties from the typekey:

Property	Description	Context to use	Example
`Code`	A non-localized `String` representation that represents this typekey. Typically you would use this code for exporting this typekey property to an external system. This name is not intended to be a display name, and must contain no space characters and might use abbreviations.	If comparing values, sending values to a remote system, or storing values for later comparison	`business`
`DisplayName`	A localized version of the display name, based on the current language settings.	For display to a user. This might include sending mail or other notifications in the local language.	`Affaires`
`UnlocalizedName`	The unlocalized version of the display name. This does not vary on the current language settings.	Only in debugging or for compatibility with earlier product releases.	`Business`
`Description`	A description of this typekey value's meaning	If you need a description (non-localized) from the data model configuration files.	`Business address type`

IMPORTANT If your application is multi-lingual and manipulates typekeys, choose very carefully whether you want to get the `DisplayName` property, the `UnlocalizedName` property, or the `Code` property. In almost all cases, use the `Code` if you might store or compare values later on or use the `DisplayName` if you are displaying something to the user. The `UnlocalizedName` property exists for debugging reasons and compatibility reasons and in general do not use it. Instead, use `Code` or `DisplayName`.

To extract display name information, you can use *`myCode`*`.DisplayName`

For example:

```
print(AddressType.TC_BUSINESS.DisplayName)
```

This code prints:

```
Business
```

chapter 3

Gosu Operators and Expressions

This topic describes the basic Gosu operators and expressions in the language.

This topic includes:

Gosu Operators

Gosu uses standard programming operators to perform a wide variety of mathematical, logical, and object manipulation operations. If you are familiar with the C, C++ or Java programming languages, you might find that Gosu operators function similar to those other languages. Gosu evaluates operators within an expression or statement in order of precedence. (For details, see "Operator Precedence" on page 62.)

Gosu operators take either a single operand (*unary* operators), two operands (*binary* operators), or three operands (a special case *ternary* operator). The following list provides examples of each operator type:

Operator type	Arguments	Examples of this operator type
unary	1	• `-3` • `typeof “Test”` • `new Array[3]`
binary	2	• `5 - 3` • `a and b` • `2 * 6`
ternary	3	• `3*3 == 9 ? true : false`

Operator Precedence

The following list orders the Gosu operators from highest to lowest precedence. Gosu evaluates operators with the same precedence from left to right. The use of parentheses can modify the evaluation order as determined by operator precedence. Gosu first evaluates an expression within parentheses, then uses that value in evaluating the remainder of the expression.

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>.
[ ]
( )</td><td>Property access, array indexing, function calls and expression grouping</td></tr>
<tr><td>new</td><td>Object creation, object reflection</td></tr>
<tr><td>,</td><td>Array value list, as in {value1, value2, value3}
Argument list, as in (parameter1, parameter2, parameter3)</td></tr>
<tr><td>as
typeas</td><td>As, typeas</td></tr>
<tr><td>+
-</td><td>Unary operands (positive, negative values)</td></tr>
<tr><td>~
!
not
typeof
eval</td><td>Bit-wise OR, logical NOT, typeof, eval(expression)</td></tr>
<tr><td>typeis</td><td>Typeis</td></tr>
<tr><td>*
/
%</td><td>Multiplication, division, modulo division</td></tr>
<tr><td><<
>>
>>></td><td>Bitwise shifting</td></tr>
<tr><td>+
-</td><td>Addition, subtraction, string concatenation</td></tr>
<tr><td><
<=
>
>=</td><td>Less than, less than or equal, greater than, greater than or equal</td></tr>
<tr><td>==
===
!=
<></td><td>Equality, inequality. For general discussion and also comparison of == and ===, see “Equality Expressions” on page 66.</td></tr>
<tr><td>&</td><td>bitwise AND</td></tr>
<tr><td>^</td><td>bitwise exclusive OR</td></tr>
<tr><td>|</td><td>bitwise inclusive OR</td></tr>
<tr><td>&&
and</td><td>Logical AND, the two variants are equivalent</td></tr>
</table>

Operator	Description
`||` `or`	Logical OR, the two variants are equivalent
`? :`	Conditional (ternary, for example, `3*3 == 9 ? true : false`)

`= += -= *=`
`/= %= &=`
`^= |= <<=`
`>>= >>>=`
	Assignment operator statements. These are technically Gosu *statements*, not *expressions*. For more information, see "Gosu Variables" on page 82.

Standard Gosu Expressions

A Gosu expression results in a single value. Expressions can be either very simple (setting a value) or quite complex. A Gosu expression is categorized by the type of operator used in constructing it. Arithmetic expressions use arithmetic operators (+, -, *, / operators) whereas logical expressions use logical operators (AND, OR, NOT operators). The following sections contain descriptions and examples of Gosu-supported expressions and how to use them.

- Arithmetic Expressions
- Conditional Ternary Expressions
- Equality Expressions
- Unary Expressions
- Evaluation Expressions
- Existence Testing Expressions
- Logical Expressions
- New Object Expressions
- Relational Expressions
- Type Cast Expressions (see "Basic Type Checking" on page 300)
- Type Checking Expressions (see "Basic Type Checking" on page 300)

Arithmetic Expressions

Gosu defines arithmetic expressions corresponding to all the common arithmetic operators, which are:

- Addition and Concatenation Operator (+)
- Subtraction Operator (-)
- Multiplication Operator (*)
- Division Operator (/)
- Arithmetic Modulo Operator (%)

Gosu supports Java big decimal arithmetic on the +, -, *, /, and % arithmetic operators. Thus, if the left- or right-hand side of the operator is a Java `BigDecimal` or `BigInteger`, then the result is `Big` also. This can be especially important if considering the accuracy, such as usually required for currency figures.

Addition and Concatenation Operator (+)

The "+" operator performs arithmetic addition or string concatenation using either two `Number` or two `String` data types as operands. The result is either a `Number` or a `String`, respectively. Note the following:

- If both operands are numeric, the "+" operator performs addition on numeric types.
- If either operand is a `String`, Gosu converts the non-`String` operand to a `String`. The result is the concatenation of the two strings.

Expression	Result
`3 + 5`	`8`
`8 + 7.583`	`15.583`

Expression	Result
`"Auto" + "Policy"`	`"AutoPolicy"`
`10 + "5"`	`"105"`
`"Number " + 1`	`"Number 1"`

Subtraction Operator (-)

The "-" operator performs arithmetic subtraction, using two `Number` values as operands.The result is a `Number`.

Expression	Result
`9 - 2`	`7`
`8 - 3.359`	`4.641`
`"9" - 3`	`6`

Multiplication Operator (*)

The "*" operator performs arithmetic multiplication, using two `Number` values as operands. The result is a `Number`.

Expression	Result
`2 * 6`	`12`
`12 * 3.26`	`39.12`
`"9" * "3"`	`27`

Division Operator (/)

The "/" operator performs arithmetic division using two `Number` values as operands. The result is a `Number`. The result of floating-point division follows the specification of IEEE arithmetic.

If either value appears to be a `String` (meaning that it is enclosed in double-quotation marks):

- If a "string" operand contains only numbers, Gosu converts the string to a number and the result is a number.
- If the "string" operand is truly a `String`, then the result is NaN (Not a Number).

Expression	Result
`10 / 2`	`5`
`5 / "2"`	`2.5`
5 / "test"	NaN
1 / 0	Infinity
0 / 0	NaN
0/1	0

Arithmetic Modulo Operator (%)

The "%" operator performs arithmetic modulo operations, using `Number` values as operands. The result is a `Number`. (The result of a modulo operation is the remainder if the numerator divides by the denominator.)

Expression	Result
`10 % 3`	`1`
`2 % 0.75`	`0.5`

Bitwise AND (&)

The “&” operator performs a binary bitwise AND operation with the value on the left side of the operator and the value on the right side of the operator.

For example, `10 & 15` evaluates to 10. The decimal number 10 is 1010 binary. The decimal number 15 is 1111 binary. In binary, this code does a bitwise AND between value 1010 and 1111. The result is binary 1010, which is decimal 10.

In contrast, `10 & 13` evaluates to 8. The decimal number 10 is 1010 binary. The decimal number 13 is 1101 binary. In binary, this does a bitwise AND between value 1010 and 1101. The result is binary 1000, which is decimal 8.

Bitwise Inclusive OR (|)

The “|” (pipe character) operator performs a binary bitwise inclusive OR operation with the value on each side of the operator.

For example, `10 | 15` evaluates to 15. The decimal number 10 is 1010 binary. The decimal number 15 is 1111 binary. In binary, this code does a binary bitwise inclusive OR with value 1010 and 1111. The result is binary 1111, which is decimal 15.

The expression `10 | 3` evaluates to 11. The decimal number 10 is 1010 binary. The decimal number 13 is 1101 binary. In binary, this does a bitwise AND between value 1010 and 1101. The result is binary 0111, which is decimal 11.

Bitwise Exclusive OR (^)

The “^” (caret character) operator performs a binary bitwise exclusive OR operation with the values on both sides of the operator.

For example, `10 | 15` evaluates to 5. The decimal number 10 is 1010 binary. The decimal number 15 is 1111 binary. In binary, this code does a binary bitwise exclusive OR with value 1010 and 1111. The result is binary 0101, which is decimal 5.

The expression `10 & 13` evaluates to 7. The decimal number 10 is 1010 binary. The decimal number 13 is 1101 binary. In binary, this does a bitwise AND between value 1010 and 1101. The result is binary 0111, which is decimal 7.

Bitwise Left Shift (<<)

The “<<” operator performs a binary bitwise left shift with the value on the left side of the operator and value on the right side of the operator.

For example, `10 << 1` evaluates to 20. The decimal number 10 is 01010 binary. In binary, this code does a binary bitwise left shift of 01010 one bit to the left. The result is binary 10100, which is decimal 20.

The expression `10 << 2` evaluates to 40. The decimal number 10 is 001010 binary. In binary, this code does a binary bitwise left shift of 001010 one bit to the left. The result is binary 101000, which is decimal 40.

Bitwise Right Shift and Preserve Sign (>>)

The “>>” operator performs a binary bitwise right shift with the value on the left side of the operator and value on the right side of the operator.

> **IMPORTANT** For signed values, the >> operator automatically sets the high-order bit with its previous value for each shift. This preserves the sign (positive or negative) of the result. For signed integer values, this is the usually the appropriate behavior. Contrast this with the `>>>` operator.

For example, `10 >> 1` evaluates to 5. The decimal number 10 is 1010 binary. In binary, this code does a binary bitwise right shift of 1010 one bit to the right. The result is binary 0101, which is decimal 5.

The expression `-10 >> 2` evaluates to -3. The decimal number -10 is 11111111 11111111 11111111 11110110 binary. This code does a binary bitwise right shift two bits to the right, filling in the top sign bit with the 1 because the original number was negative. The result is binary 11111111 11111111 11111111 11111101, which is decimal -3.

Bitwise Right Shift Right Shift and Clear Sign (>>>)

The ">>>" operator performs a binary bitwise right shift with the values on both sides of the operator.

> **IMPORTANT** The `>>>` operator sets the high-order bit with its previous value for each shift to zero. For unsigned integer values, this is the usually the appropriate behavior. Contrast this with the `>>` operator.

Equality Expressions

Equality expressions return a `Boolean` value (`true` or `false`) indicating the result of the comparison between the two expressions. Equality expressions consist of the following types:

- == Operator
- != or <> Operator

== Operator

The `==` operator tests for relational equality. The operands can be of any compatible types. The result is always `Boolean`. For reference types, Gosu language, the == operator automatically calls *`object`*`.equals()` to compare values. To compare whether the two operands are the same in-memory object, use the === operator instead. See "=== Operator Compares Object Equality" on page 66 for details.

Syntax

```
a == b
```

Examples

Expression	Result
`7 == 7`	`true`
`"3" == 3`	`true`
`3 == 5`	`false`

=== Operator Compares Object Equality

In the Java language, the `==` operator evaluates to `true` if and only if both operands have the same exact **reference value**. In other words, it evaluates to `true` if they refer to the same object in memory. This works well for primitive types like integers. For reference types, this usually is not what you want to compare. Instead, to compare *value equality*, Java code typically uses *`object`*`.equals()`, not the `==` operator.

In the Gosu language, the == operator automatically calls *`object`*`.equals()` for comparison if you use it with reference types. In most cases, this is what you want for reference types.

However, there are some cases in which you want to use identity reference, not simply comparing the values using the underlying *`object`*`.equals()` comparison. In other words, some times you want to know if two objects literally reference the same in-memory object.

You can use the Gosu operator `===` (three equals signs) to compare object equality. This always compares whether both references point to the same in-memory object.

The following examples demonstrate the difference between `==` and `===` operators:

Examples Comparing == and ===

Expression	Prints this Result	Description
`print("3" == "3")`	`true`	The two `String` objects contain the same value.
`print("3" == "4")`	`false`	The two `String` objects contain different values.
`print("3" === "4")`	`false`	Gosu represents the two `String` literals as separate objects in memory (as well as separate values).

`var x = 1 + 2`
`var s = x as String`
`print(s == "3")`
	`true`	These two variables reference the same value but different objects. If you use the double-equals operator, it returns `true`.

`var x = 1 + 2`
`var s = x as String`
`print(s === "3")`
	`false`	These two variables reference the same value but different objects. If you use the triple-equals operator, it returns `false`.

`print("3" === "3")`	`true`	This example is harder to understand. By just looking at the code, it seems like these two `String` objects would be different objects. However, in this case, the Gosu compiler is smart enough to detect they are the same `String` at compile time. Gosu optimizes the code for both usages of a `String` literal to point to the **same object** in memory for both usages of the `"3"`.

!= or <> Operator

The "!=" or "<>" operator tests for relational inequality. The operands can be of any compatible types. The result is always `Boolean`. See also the examples in "Logical NOT" on page 70 for another use of the `!=` operator.

Syntax

```
a != b
a <> b
```

Examples

Expression	Result
`7 != 7`	`false`
`"3" <> 3`	`false`
`3 <> 5`	`true`

Evaluation Expressions

The `eval()` expression evaluates Gosu source at runtime, which enables dynamic execution of Gosu source code. Gosu executes the source code within the same scope as the call to `eval()`.

Syntax

```
eval( <expression> )
```

Examples

Expression	Result
`eval( "2 + 2" )`	`4`
`eval( 3 > 4 ? true : false )`	`false`

Existence Testing Expressions

An `exists` expression iterates through a series of elements and tests for the existence of an element that matches a specific criteria.

The main way of There are two basic ways to use an `exists` expression:

- **Exists expressions in general use (outside find queries).** Gosu iterates across an array or a list but does not generate. Consider expressions like this as an alternative to simple looping with the Gosu statements `for()`, `while()`, and `do...while()`. The rest of this section focuses on this type of use.
- **For legacy-only find expressions.**

  IMPORTANT Find queries are a legacy style of querying a Guidewire database. For new code, use the query builder APIs. For details of query builder, see “Query Builder Overview” on page 129.

  If an `exists` expression in a `find` expression, Gosu optimizes the database query to use *database joins* as appropriate. Database joins are a feature of databases and query languages that push resource-intensive work and intelligence to the database if querying data from two database tables. You can think of `find` expressions as a high-level database query language embedded directly in Gosu with optimizations at compile time. For usage within `find` queries, see “Basic Find Expressions” on page 159 for more details.

If you use `exists` expressions outside `find` queries and the array or list includes entities, Gosu does not evaluate the `exists` expression using database queries. Do not rely on the exists clause to optimize the database query itself. Instead, Gosu iterates through the array until the criteria in the `where` clause is met, and then terminates.

If the array elements are cached or are otherwise in memory, the expression evaluates quickly. However, if most of the array is not readily accessible and the array is large, the task could take much processor time and real-world time. For example, if Gosu must initially load a large array from the database.

Syntax

```
exists ( [var] identifier in expression1 [index identifier] where expression2 )
```

The index variable *`identifier`* iterates through all possible array index values. The result is the type `Boolean`. The expression returns `true` to indicate success (such an element exists), or returns `false` if no such desired expression exists. If used outside a `find` expression, an `exists` expression returns `true` but does not actually save or return the desired value for later use.

Example

Expression	Result
`exists (var e in Claim.Exposures where e == null)`	`true` or `false`
`exists(var group in Claim.Exposures.AssignedGroup where group.GroupType == "autofasttrack")`	`true` or `false`

Logical Expressions

Gosu logical expressions use standard logical operators to evaluate the expression in terms of the Boolean values of `true` and `false`. Most often, logical expressions include items that are explicitly set to either `true` or `false` or evaluate to `true` or `false`. However, they can also include the following:

- `Number` values (both positive and negative numbers, regardless of their actual value) and the `String` value `"true"`, coerce to `true` if used with Boolean operators.
- `String` values **other** than the value `"true"`, which all coerce to `false` if used with Boolean operators
- the `Number` 0, which coerces to `false` if used with Boolean operators

- the value `null`, which coerces to `false` if used with Boolean operators.

IMPORTANT For important differences between `Boolean` and `boolean` types and differences in coercion, see “Boolean” on page 44.

Guidewire Gosu supports the following logical expressions:

- Logical AND
- Logical OR
- Logical NOT

As logical expressions are evaluated from left to right, they are tested for possible short-circuit evaluation using the following rules:

- `true` OR *`any-expression`* always evaluates to `true.` Gosu only runs and evaluates *`any-expression`* if the expression before the AND is true. So, if Gosu determins the expression before the `AND` evaluates to `true`, the following expression is not evaluated.
- `false` AND *`any-expression`* always evaluates to `false`. Gosu only runs and evaluates *`any-expression`* if the expression before the AND is true. So, if Gosu determins the expression before the `AND` evaluates to `false`, the following expression is not evaluated.

Logical AND

Gosu uses either `and` or `&&` to indicate a logical AND expression. The operands must be of the `Boolean` data type (or any type convertible to `Boolean`). The result is always `Boolean`.

Syntax

```
a and b
a && b
```

Examples

Expression	Result
`(4 > 3) and (3 > 2)`	(true/true) = `true`
`(4 > 3) && (2 > 3)`	(true/false) = `false`
`(3 > 4) and (3 > 2)`	(false/true) = `false`
`(3 > 4) && (2 > 3)`	(false/false) = `false`

Logical OR

Gosu uses either `or` or `||` to indicate a logical OR expression. The operands must be of the `Boolean` data type (or any type convertible to `Boolean`). The result is always `Boolean`.

Syntax

```
a or b
a || b
```

Examples

Expression	Result
`(4 > 3) or (3 > 2)`	(true/true) = `true`
`(4 > 3) || (2 > 3)`	(true/false) = `true`
`(3 > 4) or (3 > 2)`	(false/true) = `true`
`(3 > 4) || (2 > 3)`	(false/false) = `false`

Logical NOT

To indicate a logical negation (a logical NOT expression), use either the keyword `not` or the exclamation point character (`!`), also called a *bang*. The operand must be of the `Boolean` data type or any type convertible to `Boolean`. The result is always `Boolean`.

Syntax

```
not a
!a
```

Examples

Expression	Result
`!true`	`false`
`not false`	`true`
`!null`	`true`
`not 1000`	`false`

The following examples illustrate how to (not) use the NOT operator.

- **Bad example.** The following is a bad example of how to use the logical `NOT` operator.

```
if (not PolicyLine.BOPLiabilityCov.Limit ==
  PolicyLine.PolicyPeriod.MostRecentPriorBoundRevision.BOPLine.BOPLiabilityCov.Limit) {
return true
}
```

  This example causes an error if it runs because Gosu associates the `NOT` operator with the variable to its right before it evaluates the expression. In essence, the expression becomes:

```
if (false == PolicyLine.PolicyPeriod.MostRecentPriorBoundRevision.BOPLine.BOPLiabilityCov.Limit)
```

  which causes a class cast exception during the comparison, as follows:

```
'boolean (false)' is not compatible with Limit
```

- **Better example.** The following is a better example of how to use the `NOT` operator.

```
if (not (PolicyLine.BOPLiabilityCov.Limit ==
  PolicyLine.PolicyPeriod.MostRecentPriorBoundRevision.BOPLine.BOPLiabilityCov.Limit)) {
return true
}
```

  In this example, the extra parentheses force the desired comparison, then associate the `NOT` operator with it.

- **Preferred example.** Guidewire recommends that you use the following in writing code of this type.

```
if (PolicyLine.BOPLiabilityCov.Limit !=
  PolicyLine.PolicyPeriod.MostRecentPriorBoundRevision.BOPLine.BOPLiabilityCov.Limit) {
return true
}
```

  As can be seen, there was no actual need to use the `NOT` operator in this expression. The final code expression is somewhat simpler and does exactly what is asked of it.

Typeis Expressions

Gosu uses the operator `typeis` to test type information of an object. For more information, see “Basic Type Checking” on page 300.

New Object Expressions

Gosu uses the `new` operator to create an instance of a type. The type can be a Gosu class, a Java class, an array, or a Guidewire entity type.

You can use the `new` operator with any valid Gosu type, Java type, or an array. At least one constructor (creation function) must be exposed on a type to construct an instance of the type with the `new` operator.

Syntax

```
new <java-type> ( [arguments] )
new <array-type> [ <size-expression> ]
new  <array-type> [] { [array-value-list] }
```

If you pass arguments to the `new` operator, Gosu passes those arguments to the constructor. There might be multiple constructors defined, in which case Gosu uses the types and numbers of objects to choose which constructor to call.

To create Guidewire business entities such as `Claim` and `User`, for typical use do **not** pass arguments to create the entity. Passing no arguments tells Gosu to create the Guidewire entity in the current database transaction (*bundle*). This the best approach in general. Only if you are sure it is appropriate to do otherwise, pass a single argument with the `new` operator to override the bundle in which to create the new entity. For more information about bundles, see “Bundles and Transactions” on page 275.

The following table lists the behavior of the `new` operator with different parameters:

Parameter to new operator	Meaning
none	Create the entity in the current database transaction (the current *bundle*). In almost all cases, use this approach
reference to a Guidewire entity	Create the entity in the current database transaction as the entity passed as a parameter. The entity passed as a parameter must be in a non-readonly bundle.
reference to a bundle	Create the entity in the current database transaction indicated by the bundle passed directly to the `new` operator. The bundle parameter must be a non-readonly bundle.

WARNING Be extremely careful using bundle and transaction APIs, such as overriding the default behavior for bundle management. For more information about bundles, see “Bundles and Transactions” on page 275.

Examples

Expression	Result
`new java.util.HashMap(8)`	Creates an instance of the HashMap Java class.
`new String[12]`	Creates a `String` array with 12 members with no initial values.
`new String[] { "a", "b", "c" }`	Creates a `String` array with three members, initialized to “a”, “b”, and “c”.
`new Note()`	Constructs a new `Note` entity in the current bundle. Any changes related to the current code, perhaps a user change that triggered rules to run, submit to the database at the same time as the new note.
`new Note(myClaim)`	Constructs a new `Note` entity given a `Claim` as the bundle to use. If changes to the `Claim` submit to the database, the new note adds at the same time.
`new Note(myClaim.bundle)`	Constructs a new `Note` entity given a `Claim` as the bundle to use. If changes to the `Claim` submit to the database, the new note adds at the same time.

Object Initializer Syntax

Object initializers allow you to set properties on an object immediately after a `new` expression. In other words, you can assign properties as part of creating a new object. Use object initializers for compact and clear object declarations. They are especially useful if combined with data structure syntax and nested objects.

A simple version looks like the following:

```
var sampleClaim = new Claim(){ :ClaimId = "TestID" }
```

The syntax is simple: after a constructor, open a curly brace and then add pairs of property name equals values, followed by a close brace. After a constructor, open a curly and then add pairs of property name equals values. Each name/value pair has the following syntax:

```
:PROPERTY_NAME = VALUE
```

Notice that the property name has a colon before it. For multiple properties, separate multiple name/value pairs by commas.

For example, suppose you have the following code:

```
var myFileContainer = new my.company.FileContainer()
myFileContainer.DestFile = jarFile
myFileContainer.BaseDir = dir
myFileContainer.Update = true
myFileContainer.WhenManifestOnly = ScriptEnvironment.WHEN_EMPTY_SKIP
```

Instead, you can use object initializers to simplify this code to the following:

```
var myFileContainer = new my.company.FileContainer() { :DestFile = jarFile, :BaseDir = dir,
  :Update = true, :WhenManifestOnly = ScriptEnvironment.WHEN_EMPTY_SKIP }
```

Another case where this syntax is useful is naturally expressing a nested object tree, such as XML data.

For example, suppose you have the following code:

```
using xsd.test.*

var simpleTest = new SimpleTest()
simpleTest.id = "Root"

var test2 = new Test2()
test2.id = "test"

simpleTest.test2s.add(test2)
simpleTest.test2s.add(new Test2())
simpleTest.test2s.get(1).final = true
simpleTest.test2s.get(1).Test1 = new TestType()

var test1 = new xsd.test.TestType()
test1.color = Red; // Note that Gosu can infer what enum class is appropriate!
test1.number = 5

simpleTest.test4s.add(test1)
simpleTest.test3 = Blue // Since this is a simple child element, you access its value directly

return simpleTest.toXML()
```

You can instead naturally express it as:

```
using xsd.test.*

var simpleTest = new SimpleTest(){ :id = "Root", :test3 = Blue,
                                   :test2s = { new Test2(){ :id = "test" },
                                               new Test2(){ :final = true, :Test1 = new TestType() } },
                                   :test4s = { new TestType(){ :color = Red, :number = 5 } }
                                 }

return simpleTest.toXML()
```

The object initializer syntax more clearly expresses the nested nature of the XML nodes, clarifying what the generated XML looks like.

Special Syntax for Initializing Lists, Collections, and Maps

There are specialized initializer syntax and rules for creating new lists, collections, and maps, discussed in more detail in “Basic Lists” on page 231 and “Basic HashMaps” on page 233.

Referencing Existing Guidewire Entities

If you want to reference an existing Guidewire business data entity, do not use the `new` keyword.

The preferred technique is to do a query for the entity. See “Query Builder” on page 129 in the *Gosu Reference Guide* for details.

There is an alternate form to look up an entity, although not generally recommended. Type the entity name and pass the public ID as a `String` argument but omit the `new` keyword, such as:

```
var c : Claim = Claim("ABC:demo_sample:1")
```

Relational Expressions

Gosu relational operators support all types of objects that implements the `java.lang.Comparable` interface, not just numbers. Relational expressions return a `Boolean` value (`true` or `false`) indicating the result of a comparison between two expressions. Relational expressions consist of the following types:

- > Operator
- >= Operator
- < Operator
- <= Operator

It is possible to string together multiple relatational operators to compare multiple values. Add parenthesis around each individual expression. For example, the following expression ultimately evaluates to `true`:

```
( (1 <= 2) <= (3 > 4) ) >= (5 > 6)
```

The first compound expression evaluates to `false` (`(1 <= 2) <= (3 > 4)`) as does the second expression (`5 > 6`). However, the larger expression tests for greater than or equal. Therefore, as `false` is equal to `false`, the entire expression evaluates to `true`.

> Operator

The “>” operator tests two expressions for a “greater than” relationship. The operands can be either `Number`, `String`, or `DateTime` data types. The result is always `Boolean`.

Syntax

```
expression1 > expression2
```

Examples

Expression	Result
`8 > 8`	`false`
`"zoo" > "apple"`	`true`
`5 > "6"`	`false`
`currentDate > policyEffectiveDate`	`true`

>= Operator

The “>=” operator tests two expressions for a “greater than or equal” relationship. The operands can be either `Number`, `String`, or `DateTime` data types. The result is always `Boolean`.

Syntax

```
expression1 >= expression2
```

Examples

Expression	Result
`8 >= 8`	`true`
`"zoo" >= "zoo"`	`true`
`5 >= "6"`	`false`
`currentDate >= policyEffectiveDate`	`true`

< Operator

The "<" operator tests two expressions for a "less than" relationship. The operands can be either `Number`, `String`, or `DateTime` data types. The result is always `Boolean`.

Syntax

```
expression1 < expression2
```

Examples

Expression	Result
`8 < 5`	`false`
`"zoo" < "zoo"`	`false`
`5 < "6"`	`true`
`currentDate < policyEffectiveDate`	`false`

<= Operator

The "<=" operator tests two expressions for a "less than or equal to" relationship. The operands can be either `Number`, `String`, or `DateTime` data types. The result is always `Boolean`.

Syntax

```
expression1 <= expression2
```

Examples

Expression	Result
`8 <= 5`	`false`
`"zoo" <= "zoo"`	`true`
`5 <= "6"`	`true`
`currentDate <= policyEffectiveDate`	`false`

Unary Expressions

Gosu supports the following unary (single operand) expressions:

- Numeric Negation
- Typeof Expressions
- Importing Types and Package Namespaces
- Bit-wise NOT

The following sections describe these expressions. The value of a `typeof` expression cannot be fully determined at compile time. For example, an expression at compile time might resolve as a supertype. At runtime, the expression may evaluate to a more specific subtype.

Numeric Negation

Gosu uses the "`-`" operator to indicate numeric negation. The operand must be of the `Number` data type. The result is always a `Number`.

Syntax

```
-value
```

Examples

Expression	Result
`-42`	`-42`
`-(3.14 - 2)`	`-1.14`

Typeof Expressions

Gosu uses the operator `typeof` to determine meta information about an expression's type. The operand can be any valid data type. The result is the *type* of the expression. For more information, see "Basic Type Checking" on page 300.

Bit-wise NOT

The bit-wise `NOT` operator treats a numeric value as a series of bits and inverts them. This is different from the logical `NOT` operator (! — the exclamation point), which treats the entire numeral as a single Boolean value. In the following example, the logical `NOT` operator assigns a Boolean value of `true` to *x* if *y* is `false`, or `false` if *y* is `true`:

```
x = !y
```

However, in the following example, the bit-wise `NOT` operator (~ — the tilde) treats a numerical value as a set of bits and inverts each bit, including the sign operator. For example, the decimal number 7 is the binary value 0111 with a positive sign bit. If you use the bit-wise NOT, the expression ~7 evaluates to the decimal value -8. The binary value 0111 reverses to 1000 (binary value for 8), and the sign bit changes as well to -8.

Use the bit-wise `NOT` operation to manipulate a *bit mask*. A bit mask is a technique in which number or byte field maintains the state of many items where flags map to each binary digit (bit) in the field.

Importing Types and Package Namespaces

To use types and namespaces in Gosu scripts without fully qualifying the full class name including the package, use the Gosu `uses` operator. The `uses` operator behaves in a similar fashion to the Java language's `import` command, although note a minor difference mentioned later in the section. By convention, put `uses` imports at the beginning of the file or script.

While the `uses` operator is technically an unary operator in that it takes a single operand, the functionality it provides is only useful with a second statement. In other words, the only purpose of using a `uses` expression is to simplify other lines of code in which you can omit the fully-qualified type name.

Syntax

After the `uses` operator, specify a package namespace or a specific type such as a fully-qualified class name:

```
uses type
uses namespace
```

Namespaces can be specified with an asterisk (*) character to indicate a hierarchy, such as:

```
uses toplevelpackage.subpackage.*
```

Example 1

The following code uses a fully-qualified type name:

```
var map = new java.util.HashMap()
```

Instead, you can use the following code that declares an explicit type with the `uses` operator:

```
// This "uses" expression...
uses java.util.HashMap

// Use this simpler expression without specifying the full package name:
var map = new HashMap()
```

Example 2

The following code uses a fully-qualified type name:

```
var map = new java.util.HashMap()
```

Instead, you can use the following code that declares a package hierarchy with the `uses` operator:

```
// This "uses" expression...
uses java.util.*

// Use this simpler expression without specifying the full package name:
var map = new HashMap()
```

IMPORTANT Explicit types always have precedence over wildcard namespace references. This is different compared to the behavior of the Java `import` operator.

Packages Always in Scope

Some built-in Guidewire packages are always in scope, which means you do not need to use fully-qualified type names or the `uses` operator for these types. These include the packages:

- `entity.*`
- `gw.lang.*`
- `pcf.*`
- `pcftest.*`
- `perm.*`
- `productmodel.*`
- `snapshot.*`
- `soap.*`
- `typekey.*`
- `xsd.*`

Studio prevents you from creating custom packages with any of these names. For instance, if you try to create a package called `soap`, Studio displays a message in red text saying "Reserved Package".

No packages always in scope refer to Java language types. It may appear that some Java packages are always in scope because `Boolean`, `String`, `Number`, `List`, and `Object` do not require qualification. However, those do not need full package qualification because these are built-in types.

The type `List` is special in the Gosu type system. Gosu resolves it to `java.util.List` in general use but it resolves to `java.util.ArrayList` in the special case where it is used in a `new` expression. For example, the following code creates an `ArrayList` but issues a warning suggesting instead using `ArrayList`:

```
var x = new List()
```

Conditional Ternary Expressions

The conditional expression (with "`?`" and "`:`" operators) uses the `Boolean` value of one expression to decide which of two other expressions to evaluate. This logical operator is syntactically right-associative. This means that it groups right-to-left, so that `a ? b:c ? d:e ? f:g` evaluates the same as `a ? b:( c ? d:( e ? f:g ) )`. The second and third operands following the "?" must be of compatible type. (See also "Logical Expressions" on page 68 for details of Boolean logic evaluation.)

Syntax

```
expression ? result-if-true : result-if-false
```

Gosu short-circuits the evaluation of expressions in the ternary operator.

For example, given the following conditional expression:

```
condition ? expr1 : expr2
```

Gosu runs and evaluates the first expression `expr1` if and only if the `condition` expressions evaluates to `true`.

Likewise, Gosu runs and evaluates the second expression `expr2` if and only if `condition` evaluates to `false`.

Examples

Expression	Result
`3 > 4  ? true: false`	`false`
`3*3 == 9 ? true : false`	`true`

Type of the Result

If the type of the if-true clause and the if-false clause are the same, the result of the expression is that type.

However, if the true clause and the false clause return different types, the result is a combination of the types.

For example, consider the following statement:

```
var s = someConditon ? "hello" : false
```

The type of the result is the type lowest down in the type hierarchy to the types of both clauses. If either clause is a primitive type such as `int` or `boolean`, Gosu coerces the primitive type to its boxed (subclass of `Object`) version before doing this change. For example, `boolean` coerces to `Boolean`.

If they have no ancestors in common, the compile-time type of the result is `Object`. This is important to note, because it may affect the coercions you do with the result and what properties or methods you call on the result.

> **Note:** Although the compile-time type looks at both types to find a common ancestor in the type hierarchy, a ternary clause does not cause the creation of a Gosu *compound type*. For a more detailed explanation of this special type, see “Compound Types” on page 308

Special Gosu Expressions

The following sections describe various ways of working with Gosu expressions:

- Entity Literals
- Static Method Calls
- Function Calls
- Static Property Paths
- Handling Null Values

Entity Literals

An entity literal value is not the same as an entity value. It is an indirect reference to an entity with the specified ID (a sort of ID of an entity). You may cast an entity literal to the appropriate type to load the entity into memory. Entity literals are implicitly coerced to entities.

Syntax

```
<entity-type>( <entity-id> )
```

Typically, you do not create entity literals manually. The Gosu editor SmartHelp feature creates these for you as you enter code in Studio. For example, if you choose a `Group` or `User` from SmartHelp, Studio inserts an entity literal of that type into the code.

Examples

Expression	Argument meaning	Result type
`Group( "3" /* Eastern Regional Center */ )`	internal ID of a group	`Group`
`User( "34" /* Albert Lee */ )`	internal ID of a User	`User`
`ActivityPattern( "10022" /* legal_review */ )`	internal ID of an activity pattern	`ActivityPattern`

Note: See “Working in Guidewire Studio” on page 123 in the *Configuration Guide* for information on using the SmartHelp feature.

Function Calls

This expression calls a function with an optional list of arguments and returns the result.

Syntax

```
<function-name>( <argument-list> )
```

Examples

Expression	Result
`now()`	`Current Date`
`concat( "limited-", "coverage" )`	`"limited-coverage"`

Handling Null Values

As stated previously, if any object property in a path evaluates to `null`, the entire path evaluates to `null`. In essence, a `null` value anywhere along the way in a object path short-circuits evaluation and results in a `null` value, with no exception being thrown. For example, suppose that you have an expression similar to the following:

```
var groupType = claim.AssignedGroup.GroupType
```

If any element in the path evaluates to `null`, the entire expression evaluates to `null`. This includes the `claim` object also, as it is merely the first element in the path.

However, if the expression contains a method call, it may or may not cause a `NullPointerException`.

- If the method call precedes or is the `null` element, Gosu evaluates the expression as `null`.
- If a method call follows an element with a `null` value, Gosu throws a `NullPointerException`.

Example 1

Suppose that you have an expression similar to the following:

```
claim.AssignedGroup.addEvent( "abc" )
```

In this case, if either `claim` or `claim.AssignedGroup` evaluate to `null`, a `NullPointerException` is thrown as the method call follows the `null` value.

Example 2

Suppose that you have an expression similar to the following:

```
claimant.getSpecialContactRelationship().Contact.Name
```

If `Contact` is `null`, the expression evaluates to `null`. Similarly, if `getSpecialContactRelationship()` evaluates to `null`, the expression evaluates to `null`.

Example 3

Suppose that you want to test whether a user interface field—for example, a description field—is `null`. You might possibly try the following:

```
Claim.Description.length() == 0
```

In this case, if the description field is `null`, calling the method causes a `NullPointerException` exception. To avoid this situation, compare the description to `null` instead:

```
claim.Description == null
```

If you want to safely determine the `String` length, use a library function to determine the length of the field.

```
String.length( claim.Description )
```

Example 4

For those cases in which Gosu expects a Boolean value (for example, in an `if()` statement), a `null` value coerces to `false` in Gosu. This is true regardless of whether the expression's value was short-circuited. For example, the following `if()` statement's condition prints “Benefits decision not made yet”, even if `claim` or `claim.BenefitsStatusDcsn` is `null`:

```
if( not claim.BenefitsStatusDcsn ) {
  print( "Benefits decision not made yet" )
}
```

Converting Null Values with the Math.Nz() Utilities

The Gosu math utility classes contains methods that you can use to conveniently convert `null` values to 0 or to a specified value. To use these APIs, call the `Nz` methods on the `gw.api.util.Math` class.

For example:

```
var m = gw.api.util.Math

m.Nz( null )      // == 0
m.Nz( 8 )         // == 8

var big = new java.math.BigDecimal( 8 )
m.Nz( big )                // == big
m.Nz( null, NaN )          // == NaN
```

Static Method Calls

Gosu uses the following syntax to call a static method on a type.

Syntax

```
<type-expression>.<static-method-name>( <argument list> )
```

Examples

Expression	Result
`Person.isAssignableFrom( type )`	`true/false`
`java.lang.System.currentTimeMillis()`	Current time
`java.util.Calendar.getInstance()`	Java Calendar

For more information about static methods and the `static` operator, see “Modifiers” on page 178

Static Property Paths

Gosu uses the dot-separated path rooted at a `Type` expression to retrieve a static property’s value.

Syntax

```
<type-expression>.<static-property>
```

Examples

Expression	Result
`Claim.TypeInfo`	`Claim typeInfo`
`LossCause.TC_HAIL`	`"hail" typekey`
`java.util.Calendar.FRIDAY`	`Friday value`

For more information about static properties in classes, see “Modifiers” on page 178.

Entity and Typekey Type Literals

Gosu supports referencing an `entity` or `typekey` type by relative name, or by fully qualified name.

Notes:

- A fully qualified entity name begins with “`entity.`”.
- A fully qualified typekey begins with “`typekey.`”.

Syntax

```
[entity.]<type-name>
[typekey.]<type-name>
```

Examples

Expression	Result
`Claim`	`Claim` type
`entity.Claim`	`Claim` type
`LossCause`	`LossCause` type
`typekey.LossCause`	`LossCause` type
`java.lang.String`	`String` type
`int`	`int` type

chapter 4

Statements

This topic describe important concepts in writing more complex Gosu code to perform operations required by your business logic.

This topic includes:

Gosu Statements

A Gosu expression has a value, while Gosu statements do not. Between those two choices, if it is possible to pass the result as an argument to a function, then it is an expression. If it is not possible, then it is a statement.

For example, the following are all Gosu expressions as each results in a value:

```
5 * 6
typeof 42
exists ( var e in Claim.Exposures where e == null )
```

The following are all Gosu statements:

```
print(x * 3 + 5)
for (i in 10) { ... }
if( a == b ) { ... }
```

Note: Do not confuse *statement lists* with *expressions* or *Gosu blocks*. Blocks are anonymous functions that Gosu can pass as objects, even as objects passed as function arguments. For more information, see "Gosu Blocks", on page 213.

Statement Lists

A statement list is a list containing zero or more Gosu statements beginning and ending with curly braces "{" and "}", respectively.

Guidewire strongly recommends omitting semicolon characters in Gosu at the end of lines because code is more readable without optional semicolons. However, you type multiple statement lists on one line, such as within block definitions, use semicolon characters to separate statements. (For other style guidelines, see “General Coding Guidelines” on page 333.)

Syntax

```
{ statement-list }
```

Multi-line Example (No semicolons)

```
{
  var x = 0
  var y = myfunction( x )

  print( y )
}
```

Single-line Example (Semicolons)

```
var adder = \ x : Number, y : Number -> { print("I added!"); return x + y; }
```

Gosu Variables

To create and assign variables, consider the type of the variable as well as its value.

- Variable Type Declaration
- Variable Assignment

Variable Type Declaration

If a type is specified for a variable, the variable is considered strongly typed, meaning that a type mismatch error results if an incompatible value is assigned to the variable. Similarly, if a variable is initialized with a value, but no type is specified, the variable is strongly typed to the type of the value. The only way to declare a variable without a strong type is to initialize it with a `null` value without a type specified. Note, however, the variable takes on the type of the first non-`null` value assigned to it.

Syntax

```
var identifier [ : type-literal ] = expression
var identifier : type-literal [ = expression ]
```

Examples

```
var age = 42
var age2 : Number
var age3 : Number = "42"
var c : Claim
...
```

Variable Assignment

Gosu uses the standard programming assignment operator = to assign the value on the right-side of the statement to the item on the left-side of the statement.

Syntax

```
variable = expression
```

Examples

```
count = 0
time = now()
```

Gosu also supports compound assignment operators that perform an action and assign a value in one action. The following lists each compound operator and its behavior. The examples assume the variables are previous declared as `int` values.

Operator	Description	Examples
=	Simple assignment to the variable on the left-hand side of the operator with the value on the right-hand side.	`i = 10` **Assigns value 10.**

+=	Increases the value of the variable by the amount on the right-hand side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.	`i = 10`
		`i += 3`
		Assigns value 13.

-=	Increases the value of the variable by the amount on the right-hand side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.	`i = 10`
		`i -= 3`
		Assigns value 7.

*=	Multiplies the value of the variable by the amount on the right-hand side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.	`i = 10`
		`i *= 3`
		Assigns value 30.

/=	Divides the value of the variable by the amount on the right-hand side of the operator.	`i = 10`
		`i /= 3`
		Assigns value 3.
		For the `int` type, there is no fraction. If you used a floating-point-ing type, the value would be 3.333333.

%=	Divides the value of the variable by the amount on the right-hand side of the operator, and returns the remainder. Next, Gosu assigns this result to the variable on the left-hand side.	`i = 10`
		`i %= 3`
		Assigns value 1.
		This is 10 - (3.3333 as int)*3

&=	Performs a bitwise AND operation with the original value of the variable and value on the right-hand side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.	`i = 10`
		`i &= 15`
		Assigns value 10.
		The decimal number 10 is 1010 binary. The decimal number 15 is 1111 binary. This code does a bitwise AND between value 1010 and 1111. The result is binary 1010, which is decimal 10.
		Contrast with this example:
		`i = 10`
		`i &= 13`
		Assigns value 8.
		The decimal number 10 is 1010 binary. The decimal number 13 is 1101 binary. This does a bitwise AND between value 1010 and 1101. The result is binary 1000, which is decimal 8.

<table>
<tr><th>Operator</th><th>Description</th><th>Examples</th></tr>
<tr><td>^=</td><td>Performs a bitwise exclusive OR operation with the original value of the variable and value on the right side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.</td><td><code>i = 10</code>
<code>i ^= 15</code>
Assigns value 5.
The decimal number 10 is 1010 binary. The decimal number 15 is 1111 binary. This code does a bitwise exclusive OR with value 1010 and 1111. The result is binary 0101, which is decimal 5.
Contrast with this example:
<code>i = 10</code>
<code>i ^= 13</code>
Assigns value 7.
The decimal number 10 is 1010 binary. The decimal number 13 is 1101 binary. This does a bitwise AND between value 1010 and 1101. The result is binary 0111, which is decimal 7.</td></tr>
<tr><td>|=</td><td>Performs a bitwise inclusive OR operation with the original value of the variable and value on the right side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.</td><td><code>i = 10</code>
<code>i |= 15</code>
Assigns value 15.
The decimal number 10 is 1010 binary. The decimal number 15 is 1111 binary. This code does a bitwise inclusive OR with value 1010 and 1111. The result is binary 1111, which is decimal 15.
Contrast with this example:
<code>i = 10</code>
<code>i |= 3</code>
Assigns value 11.
The decimal number 10 is 1010 binary. The decimal number 13 is 1101 binary. This does a bitwise AND between value 1010 and 1101. The result is binary 0111, which is decimal 11.</td></tr>
<tr><td><<=</td><td>Performs a bitwise left shift with the original value of the variable and value on the right side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.</td><td><code>i = 10</code>
<code>i <<= 1</code>
Assigns value 20.
The decimal number 10 is 01010 binary. This code does a bitwise left shift of 01010 one bit to the left. The result is binary 10100, which is decimal 20.
Contrast with this example:
<code>i = 10</code>
<code>i <<= 2</code>
Assigns value 40.
The decimal number 10 is 001010 binary. This code does a bit-wise left shift of 001010 one bit to the left. The result is binary 101000, which is decimal 40.</td></tr>
</table>

Operator	Description	Examples

>>=	Performs a bitwise right shift with the original value of the variable and value on the right side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.
	IMPORTANT: for signed values, this operator automatically sets the high-order bit with its previous value for each shift. This preserves the sign (positive or negative) of the result. For signed integer values, this is the usually the appropriate behavior. Contrast this with the >>>= operator.
		`i = 10`
		`i >>= 1`
		Assigns value 5.
		The decimal number 10 is 1010 binary. This code does a bitwise right shift of 1010 one bit to the right. The result is binary 0101, which is decimal 5.
		Contrast with this example:
		`i = -10`
		`i >>= 2`
		Assigns value -3.
		The decimal number -10 is 11111111 11111111 11111111 11110110 binary. This code does a bitwise right shift two bits to the right, filling in the top sign bit with the 1 because the original number was negative. The result is binary 11111111 11111111 11111111 11111101, which is decimal -3.

>>>=	Performs a bitwise right shift with the original value of the variable and value on the right side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.
	IMPORTANT: this operator sets the high-order bit with its previous value for each shift to zero. For unsigned integer values, this is the usually the appropriate behavior. Contrast this with the >>= operator.
		`i = 10`
		`i >>>= 1`
		Assigns value 5.
		The decimal number 10 is 1010 binary. This code does a bitwise right shift of 1010 one bit to the right. The result is binary 0101, which is decimal 5.
		Contrast with this example:
		`i = -10`
		`i >>>= 2`
		Assigns value 1073741821.
		The negative decimal number -10 is 11111111 11111111 11111111 11110110 binary. This code does a bitwise right shift two bits to the right, with no filling of the top bit. The result is binary 00111111 11111111 11111111 11111101, which is decimal 1073741821. The original was a negative number, but in this operator that bit value is filled with zeros for each shift.

++ unary operator	Adds one to the current value of a variable. Also known as the increment-by-one operator. The unary ++ and -- operators must always appear after the variable name
	IMPORTANT: See related information in "Compound Assignment Compared to Expressions" on page 85.
		`i = 10`
		`i++`
		Assigns value 11.

-- unary operator	Subtracts one from the current value of a variable. Also known as the decrement-by-one operator. The unary ++ and -- operators must always appear after the variable name
	IMPORTANT: See related information in "Compound Assignment Compared to Expressions" on page 85.
		`i = 10`
		`i--`
		Assigns value 9.

Compound Assignment Compared to Expressions

The table above lists a variety of compound assignment operators, such as ++, --, and +=.

It is important to note that these operators form *statements*, rather than *expressions*.

This means that the following Gosu is valid

```
while(i < 10) {
  i++
  print( i )
}
```

However, the following Gosu is invalid because statements are impermissible in an *expression*, which Gosu requires in a `while` statement:

```
while(i++ < 10) { // Compilation error!
  print( i )
}
```

It is important to understand that Gosu supports the increment and decrement operator only **after** a variable, **not before** a variable. In other words, `i++` is valid but `++i` is invalid. The `++i` form exists in other languages to support expressions in which the result is an expression that you pass to another statement or expression. As mentioned earlier, in Gosu these operators do not form an expression. Thus you cannot use increment or decrement in `while` declarations, `if` declarations, and `for` declarations. Because the `++i` style exists in other languages to support forms that are *unsupported* in Gosu, Gosu does not support the `++i` form of this operator.

IMPORTANT Gosu supports the `++` operator after a variable, such as `i++`. Using it before the variable, such as `++i` is unsupported and generates compiler errors.

Gosu Conditional Execution and Looping

Gosu uses the following constructions to perform program flow:

- If() … Else() Statements
- For() Statements
- While() Statements
- Do…While() Statements
- Switch() Statements

If() … Else() Statements

The most commonly used statement block within the Gosu language is the `if()` block. The `if()` block uses a multi-part construction. The `else()` block is optional.

Syntax

```
if ( <expression> ) <statement>
[ else <statement> ]
```

Example

```
if( a == b ) { print( "a equals b" ) }

if( a == b || b == c ) { print( "a equals b or b equals c" ) }
else { print( "a does not equal b and b does not equal c" ) }

if( a == b ) { print( "a equals b" ) }
else if( a == c ) { print( "a equals c" ) }
else { print( "a does not equal b, nor does it equal c" ) }
```

To improve the readability of your Gosu code, Gosu automatically downcasts after a `typeis` expression if the type is a subtype of the original type. This is particularly valuable for `if` statements and similar Gosu structures. Within the Gosu code bounded by the `if` statement, you do not need to do casting (`as` *`TYPE`* expressions) to that subtype. Because Gosu confirms that the object has the more specific subtype, Gosu implicitly considers that variable's type to be the **subtype**, at least within that block of code. For details, see "Basic Type Checking" on page 300

For() Statements

> **Note:** Gosu provides backwards compatibility for the use of the `foreach(...)` statement, although Guidewire recommends that you use the `for(...)` statement instead.

The `for(...in...)` statement block uses a multi-part construction.

Syntax

```
for ( <identifier> in <expression> [ index <identifier> ] ) { <statement> }
```

The scope of the `<identifier>` is limited to the statement block itself. The `<expression>` in the `in` clause must evaluate to one of the following:

- `Array`
- Java `List` (or any Java collection)
- Java `Iterator`
- `String` (as a list of characters)

See also "Java-based Lists as Arrays" on page 54 for details on using lists as arrays and accessing list members using array notation.

Iteration in For() Statements

There are three ways that you can iterate through the members of the list or array contained in the `for()` statement:

- Automatic Iteration
- Automatic Iteration with Index
- Iterator Method Iteration

Automatic Iteration

Use automatic iteration to iterate automatically through the array or list members. Iteration starts with the initial member and continues sequentially until terminating at the last member. Specify this type of iteration by using the following syntax:

```
for ( member in OBJ)
```

In this case, *`OBJ`* must be a list, an array, or an integer. If it is an integer, Gosu iterates through the list that many times, and the variable contains the zero-based index.

Example:

```
for( i in 100 )
for( property in Claim.TypeInfo.Properties )
```

Automatic Iteration with Index

Use index iteration if you need to determine the exact position of a particular element of an array or list. This technique adds an explicit index to determine the index value or to access members of the array or list in a non-sequential fashion using array notation. Specify this type of iteration by using the following syntax:

```
for ( member in OBJ index loopcount)
```

Example:

```
//This example prints the index of the highest score in an array of test scores.
//This particular example prints "3".

var testScores = new Number[] {91, 75, 97, 100, 89, 99}
print( getIndexOfHighestScore( testScores ) )

function getIndexOfHighestScore( scores : Number[] ) : Number {
  var highIndex = 0

  for( score in scores index i ) {
    if( score > scores[highIndex] ) { highIndex = i }
```

```
  }

  return highIndex

}

//Result
3
```

Iterator Method Iteration

Use this type of iteration if the object over which you are iterating is **not** a list or array, but it has an iterator. See “Using the Results of Find Expressions (Using Query Objects)” on page 164 for additional iteration examples using the `iterator()` method. Specify this type of iteration by using the following syntax:

```
for(member in object.iterator() )
```

Example

```
//This example iterates over the color values in a map
var mapColorsByName = new java.util.HashMap()

mapColorsByName.put( new java.awt.Color( 1, 0, 0 ), "red" )
mapColorsByName.put( new java.awt.Color( 0, 1, 0 ), "green" )
mapColorsByName.put( new java.awt.Color( 0, 0, 1 ), "blue" )

for( color in mapColorsByName.values().iterator() ) {
  print( color )
}

//Result
red
green
blue
```

Examples

The following examples illustrate the different methods for iterating through the members of an array or list in a `for()` block.

```
//Example 1: Prints all the letters  with the index
for( var letter in gw.api.util.StringUtil.splitWhitespace( "a b c d e" ) index i ) {
  print( "Letter " + i + ": " + letter )
}

//Example 2: Print a message for the first exposure with 'other coverage'
//OtherCoverage is a Boolean property
for( var exp in Claim.Exposures ) {
  if( exp.OtherCoverage ) {
     print( "Found an exposure with other coverage." )
    // Transfer control to statement following this for…in statement
    break
  }
}

// Example 3: Prints numbers 0 - 99 using simple iteration
for( i in 100 ) {
  print( i + " of 100" )
}

// Example 4: Prints all Claim properties using reflection
for( property in Claim.TypeInfo.Properties ) {
  print( property )
}
```

While() Statements

Gosu evaluates the `while()` expression, and uses the Boolean result (it must evaluate to `true` or `false`) to determine the next course of action:

- If the expression is initially `true`, Gosu executes the statements in the statement block repeatedly until the expression becomes `false`. At this point, Gosu exits the `while` statement and continues statement execution at the next statement after the `while()` statement.

- If the expression is initially `false`, Gosu never executes any of the statements in the statement block, and continues statement execution at the next statement after the `while()` statement.

Syntax

```
while( <expression> ) {
  <statements>
}
```

Example

```
// Print the digits
var i = 0

while( i < 10 ) {
  print( i )
  i = i + 1
}
```

Do...While() Statements

The `do...while()` block is similar to the `while()` block in that it evaluates an expression and uses the `Boolean` result to determine the next course of action. The principal difference, however, is the Gosu tests the expression for validity **after** executing the statement block, instead of prior to executing the statement block. This means that the statements in the statement block executes at least once (initially).

- If the expression is initially `true`, Gosu executes the statements in the statement block repeatedly until the expression becomes `false`. At this point, Gosu exits the `do...while()` block and continues statement execution at the next statement after the `do...while()` statement.
- If the expression is initially `false`, Gosu executes the statements in the statement block once, then evaluates the condition. If nothing in the statement block has changed so that the expression still evaluates to `false`, Gosu continues statement execution at the next statement after the `do...while()` block. If action in the statement block causes the expression to evaluate to `true`, Gosu executes the statement block repeatedly until the expression becomes `false`, as in the previous case.

Syntax

```
do {
  <statements>
} while( <expression> )
```

Example

```
// Print the digits
var i = 0

do {
  print( i )
  i = i + 1
} while( i < 10 )
```

Switch() Statements

Gosu evaluates the `switch()` expression, and uses the result to choose one course of action from a set of multiple choices. Gosu evaluate the expression, then iterates through the case expressions in order until it finds a match.

- If a case value equals the expression, Gosu execute its accompanying statement list. Statement execution continues until Gosu encounters a `break` statement, or the switch statement ends. Gosu continues to the next case (Gosu executes multiple case sections) if you omit the `break` statement.
- If no case value equals the expression, Gosu skips to the default case, if one exists. The default case is a case section with the label `default:` rather than `case` *`VALUE`*`:`. The default case must be the last case in the list of sections.

The `switch()` statement block uses a multi-part construction. The `default` statement is optional. However, in most cases, it is best to implement a default case to handle any unexpected conditions.

Syntax

```
switch( <expression> ) {
  case label1 :
    [statementlist1]
    [break]
  [ ...
  [ case labelN :
    [statementlistN]
    [break] ] ]
  [ default :
    [statementlistDefault]]
}
```

Example

```
switch( strDigitName ) {
  case "one":
    strOrdinalName = "first"
    break
  case "two":
    strOrdinalName = "second"
    break
  case "three":
    strOrdinalName = "third"
    break
  case "five":
    strOrdinalName = "fifth"
    break
  case "eight":
    strOrdinalName = "eighth"
    break
  case "nine":
    strOrdinalName = "ninth"
    break
  default:
    strOrdinalName = strDigitName + "th"
}
```

To improve the readability of your Gosu code, Gosu automatically downcasts the object after a `typeis` expression if the type is a subtype of the original type. This is particularly valuable for `if` statements and similar Gosu structures such as `switch`. Within the Gosu code bounded by the `if` or `switch` statement, you do not need to do casting (`as` *`TYPE`* expressions) to that subtype for that case. Because Gosu confirms that the object has the more specific subtype, Gosu implicitly considers that variable's type to be the **subtype** for that block of code. There are several special cases that turn off the downcasting. For details, see "Basic Type Checking" on page 300.

Gosu Functions

Functions encapsulate a series of Gosu statements to perform an action and optionally return a value. Generally speaking, functions exist attached to a type (for example, functions declared within a class). As in other object-oriented languages, functions are also referred to as *methods*.

> **Note:** The built-in `print` function is an exception to this rule. The `print` function is always in scope, and is not attached to a type.

Gosu does not support functions defined within other functions. However, you can use the Gosu feature called blocks to do something similar. See "What Are Blocks?", on page 213 for more information.

Unlike Java, Gosu does not support variable argument functions (so-called `vararg` functions), meaning that Gosu does not support arguments with "`...`" arguments.

Gosu permits you to specify only type literals for a function's return type. Guidewire does not support other expressions that might evaluate (indirectly) to a type.

Gosu requires that you provide the return type in the function definition, unless the return type is `void` (no return value). If the return type void, omit the type and the colon before it. Also, any `return` statement must return a type that matches the declared function return type. A missing return type or a mismatched return value generates a compiler error.

Syntax

```
[modifiers] function IDENTIFIER ( argument-declaration-list ) [:type-literal] {
  function-body
}
```

Examples

```
function square( n : Number ) : Number {
  return n * n
}

// Compile error "Cannot return a value from a void function."
private function myfunction() {
  return "test for null value"
}

function fibonacci( n : Number ) : Number {
  if (n == 0) { return 0 }
  else if (n == 1 ) { return 1 }
  else {return fibonacci( n - 1 ) + fibonacci( n - 2 ) }
}

function concat ( str1:String, str2:String ) : String {
  return str1 + str2
}
```

For more information about modifiers that can appear before the word function in class definitions, see "Modifiers" on page 178.

If the return type is not `void`, **all** possible code paths must return a value in a method that declares a return type. In other words, if any code path contains a `return` statement, Gosu requires a `return` statement for all possible paths through the function. The set of all paths includes all outcomes of conditional execution, such as `if` and `switch` statements.

For example, the following method is invalid:

```
//invalid...
class MyClass {
  function myfunction(myParameter) : boolean {
    if myParameter==1
      return true
    if myParameter==2
      return false
  }
}
```

Gosu generates a "Missing Return Statement" error for this function and you must fix this error. The Gosu compiler sees two separate `if` expressions for a total of four total code paths. Even if you believe the function is always used with *`myParameter`* set to value 1 or 2 but no other value, you must fix the error. To fix the error, rewrite the code so that all code paths contain a `return` statement.

For example, the earlier example can be fixed using an `else` clause:

```
class MyClass {
  function myfunction(myParameter) : boolean {
    if myParameter==1
      return true
    else
      return false
  }
}
```

Similarly, if you use a `switch` statement, consider using an `else` section.

This strict requirement for `return` statements mirrors the analogous requirements in the Java language.

Public and Private Functions

A function is public by default, meaning that it can be called from anywhere in a Guidewire application that uses Gosu. In contrast, a private function can be called only within the library in which it is defined. For example, suppose you have the following two functions defined in a library:

```
public function funcA() {
  ...
```

```
}

private function funcB() {
  ...
}
```

Because funcA() is defined as public, it can be called from any other Gosu expression. However, funcB() is private, and therefore is not valid anywhere except within the library.

For example, a function in another library could call funcA(), but it could not call the private funcB(). Because funcA() is defined in the same library as funcB(), however, funcA() can call funcB().

Guidewire recommends that you do **not** make any function public without good reason. Therefore, mark a function as private if it is defined only for use inside the library.

Note: See "Modifiers" on page 178 for more information on class and function level access modifiers.

chapter 5

Exception Handling

Gosu supports the following standard exception handling constructions from other languages such as throw statements, try/catch/finally blocks, and special Gosu statements such using keyword.

This topic includes:

Try-Catch-Finally Constructions

The `try...catch...finally` blocks provides a way to handle some or all of the possible errors that may occur in a given block of code during runtime. If errors occur that the script does not handle, Gosu simply provides its normal error message, as if there was no error handling.

The `try` block contains code where an error can occur, while the `catch` block contains the code to handle any error that does occur.

- If an error occurs in the `try` block, Gosu passes program control to the `catch` block for processing. The initial value of the error-identifier is the value of the error that occurred in the `try` block.
- If an error is thrown from Java code, the value is the exception or error that was thrown. Otherwise, the value is an exception thrown elsewhere in Gosu code.
- If no error occurs, Gosu does not execute the `catch` block.
- If the error cannot be handled in the `catch` block associated with the `try` block where the error occurred, use the `throw` statement. The `throw` statement rethrows the exception to a higher-level error handler.

After all statements in the `try` block have been executed and any error handling has occurred in the `catch` block, the `finally` block is unconditionally executed.

Gosu executes the code inside the `finally` block, even if a `return` statement occurs inside the `try` or `catch` blocks, or if an error is thrown from a `catch` block. Thus, Gosu guarantees that the `finally` block executes.

Syntax

```
try
  <try statements>
[catch( exception )
  <catch statements>]
[finally
  <finally statements>]
```

Example

```
try {
  print( "Outer TRY running..." )
  try {
    print( "Nested TRY running..." )
    throw "an error"
  }
  catch( e ) {
    print( "Nested CATCH caught "+e )
    throw e + " rethrown"
  }
  finally { print( "Nested FINALLY running..." ) }
}
catch( e ) { print( "Outer CATCH caught " + e ) }
finally { print( "Outer FINALLY running" ) }
```

Output

```
Outer TRY running...
Nested TRY running...
Nested CATCH caught an error
Nested FINALLY running...
Outer CATCH caught an error rethrown
Outer FINALLY running
```

Throw Statements

The `throw` statement generates an error condition which you can handle through the use of `try…catch…finally` blocks.

> **Note:** Guidewire strongly recommends that you do **not** use `throw` statements as part of regular program execution, but rather for dealing with true error conditions only.

Syntax

```
throw <expression>
```

In the following examples, notice how the error message changes if the value of x changes from 0 to 1.

Example 1

```
try {

  var x = 0

  try {
    if( x == 0 ) { throw "x equals zero" }
    else { throw "x does not equal zero" }
  } catch( e ) {
    if( e == "x equals zero" ) { return( e + " handled locally." ) }
    else { throw e }
  }
} catch( e ) { return( e + " handled higher up." ) }

//Output
x equals zero handled locally.
```

Example 2

```
try {

  var x = 1

  try {
    if( x == 0 ) { throw "x equals zero" }
```

```
    else { throw "x does not equal zero" }
  } catch( e ) {
    if( e == "x equals zero" ) { return( e + " handled locally." ) }
    else { throw e }
  }
} catch( e ) { return( e + " handled higher up." ) }

//Output
x does not equal zero handled higher up.
```

Checked Exceptions in Gosu

Gosu allows you to catch and test for catch *checked exceptions*. Checked exceptions identify **specific** types of problems, typically thrown by Java code at some lower level. At a fundamental level, Gosu does not natively distinguish between checked and unchecked exceptions. Generally speaking, use standard unchecked exceptions for designing new code and APIs. However, the language provides a simplified syntax for catching checked exceptions.

The standard syntax for `catch` is simply: `catch(e)` with no declared type for `e` although the exception has type:

```
com.guidewire.commons.gosu.parser.statements.ThrowStatement.GosuThrowException
```

Gosu natively represents *checked* exceptions as an object hanging off of the caught exception in its `Cause` property containing an object of type `Throwable`. The `Cause` property is `null` if the exception has no checked exception associated with it.

The class `Throwable` is the superclass of all errors and exceptions in the Java language. Use the two subclasses `Error` and `Exception` to indicate exceptional situations occurred. Your code can understand the exception by its subclass of those classes, such as the Java `IOException` or `NoSuchMethodException` subclasses.

For example, the following Gosu code catches any specific `IOException` objects stored in the `Cause` property within the exception:

```
try {
  doSomethingThatMayThrowIOException()
}
Catch( e ) {
  if( e.Cause typeis IOException )
  {
    // Handle the IOException, which is the only type this catches
  }
  else
  {
    // rethrow the exception if it is not the right type
    throw e
  }
}
```

However, Gosu provides a concise syntax that lets you catch only specific checked exceptions in an approach similar to Java's `try`/`catch` syntax. Simply declare the exception of the type of exception you wish to catch:

```
catch( e : ThrowableSubclass )
```

For example:

```
try {
  doSomethingThatMayThrowIOException()
}
catch( e : IOException ) {
  // Handle the IOException
}
```

IMPORTANT Guidewire recommends you not use checked exceptions in Gosu code. However, if you definitely need to handle a specific exception, Guidewire suggests you use this new simplified syntax to make your code more readable.

Add a `finally` block at the end to perform cleanup code that runs for errors and for success code paths:

```
try {
  doSomethingThatMayThrowIOException()
```

```
}
catch( e : IOException ) {
}
finally {
  // PERFORM CLEANUP HERE
}
```

Object Lifecycle Management ('using' Clauses)

If you have an object with a lifecycle of a finite extent of code, you can simplify your code with the new `using` statement. The `using` statement is a more compact and less error-prone way of working with resources than using `try`/`catch`/`finally` clauses. The cleanup always occurs without requiring a separate `finally` clause, nor do you need to explicitly check whether resources have `null` values. The `using` statement also simplifies synchronization and locking, discussed more later in this section.

For example, to use an output stream typically code would open the stream, then use it, then close it to dispose of related resources. If something goes wrong while using the output stream, your code must close the output stream and perhaps check whether it successfully opened before closing it. In Gosu (or Java) you can use a `try`/`finally` block like the following to clean up the stream:

```
OutputStream os = SetupMyOutputStream() // insert your code that creates your output stream
try {
  //do something with the output stream
}
  finally {
  os.close();
}
```

You can simplify your code using the Gosu `using` statement as follows:

```
using( var os = SetupMyOutputStream() ) {

     //do something with the output stream

} // Gosu disposes of the stream after it completes or if there is an exception
```

The basic form of a using clause is as follows:

```
using( ASSIGNMENT_OR_LIST_OF_STATEMENTS )
{
  // do something here
}
```

The parentheses after the `using` keyword can contain either a Gosu expression or a comma-delimited list of one or more Gosu statements. Gosu runs any statements (including variable assignment) at run time and uses the result as an object to manage in the `using` clause.

> **Note:** You do not need an additional `return` statement to pass the value to the `using` clause. Also note that the statements must be delimited with commas, not semicolons.

There are several categories of objects that work with the `using` keyword: *disposable* objects, *closeable* objects, and *reentrant* objects. If you try to use an object that does not satisfy the requirements of one of these categories, Gosu displays a compile error. The following subtopics discuss these three types of objects.

> **Note:** If Gosu detects that an object is more than one category, at run time Gosu considers the object only one category, defined by the following precedence: *disposable*, *closeable*, *reentrant*. For example, if an object has a `dispose` and `close` method, Gosu only calls the `dispose` method.

You can return values from `uses` clauses using the standard `return` statement, discussed further in "Returning Values from 'using' Clauses" on page 100.

Disposable Objects

Disposable objects are objects that Gosu can dispose to release all system resources. For Gosu to recognize a valid disposable object, the object must have one of the following attributes:

- Implements the Gosu interface called `IDisposable`, which contains only a single method called `dispose`. This method takes no arguments. Guidewire recommends always using a type that implements `IDisposable` if possible due to faster run time performance.
- Has a `dispose` method even if it does not implement the `IDisposable` interface. This approach works but is slower at run time because Gosu must use reflection (examining the type at run time) to find the method.

A type's `dispose` method must release all the resources that it owns. The `dispose` method must release all resources owned by its base types by calling its parent type's `dispose` method.

To help ensure that resources clean up appropriately even under error conditions, you must design your `dispose` method such that Gosu can call it multiple times without throwing an exception. In other words, if the stream is already closed, then invoking this method has no effect nor throw an exception.

Closeable Objects and 'using' Clauses

Closeable objects include objects such as data streams, reader or writer objects, and data channels. Many of the objects in the package `java.io` are closeable objects. For Gosu to recognize a valid closeable object, the object must have one of the following attributes:

- Implements the Java interface `java.io.ICloseable`, which contains only a single method called `close`. This method takes no arguments. Guidewire recommends using a type that implements `ICloseable` if possible due to faster run time performance.
- Has a `close` method even if it does not implement the `ICloseable` interface. This approach works but is slower at run time because Gosu must use reflection (examining the type at run time) to find the method.

A type's `close` method must release all the resources that it owns. The `close` method must release all resources owned by its base types by calling its parent type's `dispose` method.

To help ensure that resources clean up appropriately even under error conditions, you must design your `dispose` method such that Gosu can call it multiple times without throwing an exception. In other words, if the object is already closed, then invoking this method must have no effect nor throw an exception.

The following example creates a new Java file writer instance (`java.io.FileWriter`) and uses the more verbose `try` and `finally` clauses:

```
var writer = new FileWriter( "c:\\temp\\test1.txt" )
try
{
  writer.write( "I am text within a file." )
}
finally
{
  if( writer != null )
  {
    writer.close()
  }
}
```

In contrast, you can write more readable Gosu code using the `using` keyword:

```
using( var writer = new FileWriter( "c:\\temp\\test1.txt" ) )
{
  writer.write( "I am text within a file." )
}
```

You can list multiple

```
using( var reader = new FileReader( "c:\\temp\\usingfun.txt" ),
       var writer = new FileWriter( "c:\\temp\\usingfun2.txt" ) )
{
  writer.write( StreamUtil.getContent( reader ) )
}
```

JDBC Resources and Using Clauses

The following example shows how to use a using clause with a JDBC (Java Database Connection) object.

```
uses java.sql.*
```

```
...

function sampleJdbc( con : Connection )
  {
    using( var stmt = con.createStatement(),
           var rs = stmt.executeQuery( "SELECT a, b FROM TABLE2" ) )
    {
      rs.moveToInsertRow()
      rs.updateString( 1, "AINSWORTH" )
      rs.insertRow()
    }
  }
```

Reentrant Objects and 'using' Clauses

Re-entrant objects are objects that help manage safe access to data that is shared by re-entrant or concurrent code execution. For example, if you must store data that is shared by multiple threads, ensure that you protect against concurrent access from multiple threads to prevent data corruption. The most prominent type of shared data is class *static variables*, which are variables that are stored on the Gosu class itself.

For Gosu to recognize a valid reentrant object, the object must have one of the following attributes:

- Implements the `java.util.concurrent.locks.Lock` interface. This includes the Java classes in that pacakge: `ReentrantLock`, `ReadWriteLock`, `Condition`.
- Casted to the Gosu interface `IMonitorLock`. You can cast **any** arbitrary object to `IMonitorLock`. This is useful to cast Java monitor locks to this Gosu interface. For more information about monitor locks, refer to:
  `http://en.wikipedia.org/wiki/Monitor_(synchronization)`
- Implements the Gosu class `gw.lang.IReentrant`. This interface contains two methods with no arguments: `enter` and `exit`. Your code must properly lock or synchronize data access as appropriate during the `enter` method and release any locks in the `exit` method.

For blocks of code using locks (code that implements `java.util.concurrent.locks.Lock`), a `using` clause simplifies your code.

The following code uses the `java.util.concurrent.locks.ReentrantLock` class using a longer (non-recommended) form:

```
// in your class variable definitions...
var _lock : ReentrantLock = new ReentrantLock()


function useReentrantLockOld() {
  _lock.lock()
  try {
      // do your main work here
    }
  finally {
    _lock.unlock()
  }
}
```

In contrast, you can write more readable Gosu code using the `using` keyword:

```
// in your class variable definitions...
var _lock : ReentrantLock = new ReentrantLock()


function useReentrantLockNew() {
  using( _lock ) {
    // do your main work here
  }
}
```

Similarly, you can cast any object to a monitor lock by adding "`as IMonitorLock`" after the object. For example, the following method call code uses itself (using the special keyword `this`) as the monitor lock:

```
function monitorLock() {
  using( this as IMonitorLock ) {
      // do stuff
    }
  }
```

This approach effectively is equivalent to a `synchronized` block in the Java language.

Assigning Variables Inside 'using' Expression Declaration

The using clause supports assigning a variable inside the declaration of the `using` clause.

This is useful if the expression that you pass to the `using` expression is both:

- something other than a single variable
- you want to reference it from inside the statement list inside the `using` clause declaration

For example, suppose you call a method that returns a file handle and you pass that to the `using` clause as the lock. From within the `using` clause contents, you probably want to access the file so you can iterate across its contents.

To simplify this kind of code, assign the variable before the expression using the `var` keyword:

```
using ( var VARIABLE_NAME = EXPRESSION ) {
  // code that references the VARIABLE_NAME variable
}
```

For example:

```
using( var out = new FileOutputStream( this, false ) ) {
  out.write( content )
}
```

Passing Multiple Items to the 'using' Statement

You can pass multiple items in the using clause expression. Separate each item by a comma character.

For example,

```
function useReentrantLockNew() {
  using( _lock1, _lock2, _lock3) {
    // do your main work here
  }
}
```

You can combine the multiple item feature with the ability to assign variables. For more about assigning variables, see "Assigning Variables Inside 'using' Expression Declaration" on page 99 .

For example:

```
using( var lfc = new FileInputStream(this).Channel,
       var rfc = new FileInputStream(that).Channel ) {

      var lbuff  = ByteBuffer.allocate(bufferSize)
      var rbuff  = ByteBuffer.allocate(bufferSize)

      while (lfc.position() < lfc.size()) {
        lfc.read(lbuff)
        rfc.read(rbuff)

        if (not Arrays.equals(lbuff.array(), rbuff.array()))
        {
          return true
        }

        lbuff.clear()
        rbuff.clear()
      }
      return false
    }
  }
```

Gosu ensures that all objects are properly cleaned up. In other words, for each object to create or resource to acquire, if it creates or acquires successfully, Gosu releases, closes, or disposes the object. Also note that if one of the resources fails to create, Gosu does not attempt to acquire other resources in later-appearing items in the command-seperated list. Instead, Gosu simply releases the ones that did succeed.

IMPORTANT There is much more information about concurrency in the section “Concurrency” on page 311, including other concurrency APIs.

The profiler tag class (`gw.api.profiler.ProfilerTag`) implements the `IReentrant` interface. This class adds hints to the Gosu profiler which actions happen within a block of code. For details see “Using Profiler Tags” on page 100.

Using Profiler Tags

The profiler tag class (`gw.api.profiler.ProfilerTag`) hints to the Gosu profiler what actions happen within a block of code. To create a new profiler tag, pass two `String` values for the name and the description

```
new ProfilerTag("EventPayloadXML", "Calculating the event message payload XML")
```

To use the profiler tag, pass it to a `using` clause, as follows:

```
using( new ProfilerTag("EventPayloadXML", "Calculating the event message payload XML") ) {

  // do your main work here...

  // use ProfilerTag methods if desired.
  tag.setCounterValue( "test", 3 )
}
```

For more information about the profiler, see “Guidewire Profiler” on page 155 in the *System Administration Guide*.

Returning Values from ‘using’ Clauses

You can return values from within `uses` clauses using the standard `return` statement. If you return a value from within a using clause, Gosu considers the clause complete so it calls your object’s final lifecycle management method to clean up your resources. (Gosu calls the `dispose`, `close`, or `exit` method, depending on the type of object.)

The following Gosu example opens a file using the Java `BufferedReader` class and reads lines from the file until the line matches a regular expression. If code in the `while` loop finds a match, it immediately returns the value and skips the rest of the code within the `using` clause.

```
uses java.io.File
uses java.io.BufferedReader
uses java.io.FileReader

function containsText( file : File, regExp : String ) : boolean {
    using( var reader = new BufferedReader( new FileReader( file ) ) ) {
      var line = reader.readLine()
      while( line != null ) {
        if( line.matches( regExp ) ) {
          return true
        }
        line = reader.readLine() // read the next line
      }
    }
    return false
  }
}
```

chapter 6

Java and Gosu

You can write Gosu code that uses Java types. Gosu code can instantiate Java types, access properties of Java types and call methods of Java types.

If you are considering writing Java code for your Gosu to call, consider instead writing that code directly in Gosu. Remember that from Gosu, you can do everything Java can do, including directly call existing Java classes and Java libraries. You can even write Gosu code enhancements that add properties and methods to Java types, and the new members are accessible from all Gosu code.

This topic describes how to write and deploy Java code to work with Gosu, and how to call it from Gosu.

IMPORTANT This topic does **not** focus on differences between the syntax of Gosu and Java. For that information, refer to the topic “Gosu Introduction” on page 13 and “Notable Differences Between Gosu and Java” on page 36.

This topic includes:

Overview of Calling Java from Gosu

Gosu can directly use Java types as if they were native Gosu types.

Gosu can do all of the following:

- instantiate Java types
- manipulate Java objects (and primitives) as native Gosu objects.
- get variables from Java types
- call methods on Java types.

  Note: For methods that look like getters and setters, Gosu exposes methods instead as properties. For more information, see "Java Get and Set Methods Convert to Gosu Properties" on page 104.
- add new methods to Java types using Gosu *enhancements*.
- add new properties to Java types using Gosu *enhancements*. (readable, writable, or read/write)
- create Gosu classes that extend Java classes
- create Gosu interfaces that extend Java interfaces
- use Java enumerations
- use Java annotations

All of these features work with built-in Java types as well as your own Java classes and libraries. You can write Java classes that any Gosu code can call.

If you do not use Guidewire entity types, you do not need to do anything other than to put compiled classes in a special directory. In practice, most Gosu-to-Java classes have method arguments or return values that include Guidewire entity types. Because of this, you must to understand a special libraries that permits safe data transfer between Gosu and Java language: the *Java entity libraries*. The entity library interface permits your Java code to safely access nearly all methods on Guidewire entities in addition to getter and setter methods. For more information, see "Java Entity Libraries Overview", on page 114.

IMPORTANT For important details about passing arguments and return values from Gosu to Java, or from Java to Gosu, see "Java Entity Libraries Overview", on page 114.

IMPORTANT This topic does **not** focus on differences between the syntax of Gosu and Java. For that information, refer to the topic "Gosu Introduction" on page 13 and "Notable Differences Between Gosu and Java" on page 36.

Java Classes are First-Class Types

The most important thing to do know about Gosu's Java compatibility is that Java classes are first-class types in Gosu. For example, standard Java classes and custom Java classes can be instantiated with the `new` keyword:

```
var b = new java.lang.Boolean(false)
```

Many Java Classes are Core Classes for Gosu

Many core Gosu classes actually reference Java types. For example:

- the class `java.util.String` is the basic text object class for Gosu code.

- the basic collection types in Gosu simply reference the Java versions, such as `java.util.ArrayList`.

```
print(list.get(0))
```

Java Packages in Scope

Many Java packages are in scope and thus do not need fully-qualified class names or explicit “uses” statements. All types in the package `java.lang.*` are automatically in scope.

So, although you could use the code:

```
var f = new java.lang.Float(7.5)
```

The code is easier to understand with the simpler code:

```
var f = new Float(7.5)
```

Static Members in Gosu

Gosu supports *static* members (variables, functions, and property declarations) on a type. A static member means that the member exists only on the type (which exists only once), not on *instances* of the type. You can access static members on Java types just as you would native Gosu types.

For Gosu code that accesses static members, you must qualify the class that declares the static member. For example, to use the `Math` class’s cosine function and its static reference to value `PI`:

```
Math.cos(Math.PI * 0.5)
```

Gosu does not have an equivalent of the *static import* feature of Java 1.5, which allows you to omit the characters “`Math.`”.

This is only a syntax difference for using static members from Gosu (independent of whether the type is implemented in Gosu or Java). If you are writing Gosu code that calls static members of Java types, this does not affect how you write your Java code.

Simple Java Example

The following is a simple example of calling a Java class.

In your Java IDE, create and compile the following Java class called `Echo`:

```
package gw.doc.examples;

public class Echo {

  public String EchoStrings (String a, String b) {
  String modifiedA = "First Arg was " + a + "\n";
  String modifiedB = "Second Arg was " + b + "\n";
  return modifiedA + modifiedB;
  }

  public void PrintStrings (String a, String b) {
  String modifiedA = "First Arg was " + a + "\n";
  String modifiedB = "Second Arg was " + b + "\n";
  System.out.print(modifiedA + modifiedB);

   }


}
```

Next, copy the `Echo.class` file to the following location:

```
ClaimCenter/modules/configuration/plugins/gosu/classes/Echo.class
```

Next, launch Studio. If Studio is currently open, quit and relaunch it so it detects your new Java class.

Open the Gosu Tester and paste the following simple program:

```
classpath "gsrc"

var e = new gw.doc.examples.Echo()
```

```
e.PrintStrings("hello", "world")
```

If you run this Gosu code, it prints:

```
First Arg was hello
Second Arg was world
```

Similarly, you can pass data between Java and Gosu:

```
classpath "gsrc"

var e = new gw.doc.examples.Echo();

var fromJava = e.EchoStrings("hello", "world")

print (fromJava)
```

Java Get and Set Methods Convert to Gosu Properties

Gosu can call methods on Java types. For methods on Java types that look like getters and setters, Gosu exposes methods instead as properties. Gosu uses the following rules for methods on Java types:

- If the method name starts with `set` and takes exactly one argument, Gosu exposes this as a property. The property name matches the original method but without the prefix `set`. For example, suppose the Java method signature is `setName(String thename)`. Gosu exposes this a property set function for the property called `Name`.
- If the method name starts with `get` and takes no arguments and returns a value, Gosu exposes this as a getter for the property. The property name matches the original method but without the prefix `get`. For example, suppose the Java method signature is `getName()`. Gosu exposes this a `property get` function for the property named `Name` of type `String`.
- Similar to the rules for `get`, the method name starts with `is` and takes no arguments and returns a Boolean value, Gosu exposes this as a property accessor (a *getter*). The property name matches the original method but without the prefix `is`. For example, suppose the Java method signature is `isVisible()`. Gosu exposes this a `property get` function for the property named `Visible`.

If there is a setter and a getter, Gosu makes the property readable and writable. If the setter is absent, Gosu makes the property read-only. If the getter is absent, Gosu makes the property write-only.

For example, create and compile this Java class:

```
package gw.doc.examples;

public class Circle {
  public static final double PI = Math.PI;
  private double _radius;

  //Constructor #1 - no arguments
  public Circle() {
  }

  //Constructor #2
  public Circle( int dRadius ) {
  _radius = dRadius;
  }

  // from Java these are METHODS that begin with get, set, is
  // from Gosu these are PROPERTY accessors

  public double getRadius() {
    System.out.print("running Java METHOD getRadius() \n");
  return _radius;
  }
  public void   setRadius(double dRadius) {
   System.out.print("running Java METHOD setRadius() \n");
  _radius = dRadius;
  }
  public double getArea() {
   System.out.print("running Java METHOD getArea() \n");
  return PI * getRadius() * getRadius();
  }
  public double getCircumference() {
```

```
        System.out.print("running Java METHOD getCircumference() \n");
       return 2 * PI * getRadius();
       }
       public boolean isRound() {
        System.out.print("running Java METHOD isRound() \n");
       return(true);
       }


       // ** the following methods stay as methods, not properties! **

       //  For GET/IS, the method must take 0 args and return a value
       public void isMethod1 () {
       System.out.print("running Java METHOD isMethod1() \n");
       }
       public double getMethod2 (double a, double b) {
       System.out.print("running Java METHOD isMethod2() \n");
         return 1;
       }

       //  For SET, the method must take 1 args and return void
       public void setMethod3 () {
        System.out.print("running Java METHOD setMethod3() \n");
       }
       public double setMethod4 (double a, double b) {
        System.out.print("running Java METHOD setMethod4() \n");
        return 1;
       }

     }
```

The following Gosu code uses this Java class. Note which Java methods become property accessors and which ones do not.

```
// instatiate the class with the constructor that takes an argument
var c = new gw.doc.examples.Circle(10)

// Use natural property syntax to SET GOSU PROPERTIES. In Java, this was a method.
c.Radius = 10

// Use natural property syntax to GET GOSU PROPERTIES
print("Radius " + c.Radius)
print("Area " + c.Area)
print("Round " + c.Round) // boolean true  coerces to String "true"
print("Circumference " + c.Circumference)

// the following would be syntax errors if you uncomment. They are not writable (no setter method)
// c.Area = 3
// c.Circumference = 4
// c.Round = false

// These Java methods do not convert to properties (wrong number of arguments or wrong type)
c.isMethod1()
var temp2 = c.getMethod2(1,2)
c.setMethod3()
var temp4 = c.setMethod4(8,9)
```

This Gosu code outputs the following:

```
running Java METHOD setRadius()
running Java METHOD getRadius()
Radius 10
running Java METHOD getArea()
running Java METHOD getRadius()
running Java METHOD getRadius()
Area 314.1592653589793
running Java METHOD isRound()
Round true
running Java METHOD getCircumference()
running Java METHOD getRadius()
Circumference 62.83185307179586
running Java METHOD isMethod1()
running Java METHOD isMethod2()
running Java METHOD setMethod3()
running Java METHOD setMethod4()
```

Interfaces

Gosu classes can directly implement Java interfaces.

Gosu interfaces can directly extend Java interfaces.

Enumerations

Gosu can directly use Java enumerations.

Annotations

Gosu can directly use Java annotations.

Java Primitives

Gosu supports the following primitive types: `int`, `char`, `byte`, `short`, `long`, `float`, `double`, `boolean`, and the special value that means an empty object value: `null`. This is the full set that Java supports, and the Gosu versions are fully compatible with the Java primitives, in both directions.

Additionally, every Gosu primitive type (other than the special value `null`) has an equivalent object type defined in Java. This is the same as in Java. For example, for `int` there is the `java.lang.Integer` type that descends from the `Object` class. This category of object types that represent the equivalent of primitive types are called *boxed primitive* types. In contrast, primitive types are also called *unboxed primitives*. In most cases, Gosu converts between boxed and unboxed primitive as needed for typical use. However, they are slightly different types, just as in Java, and on rare occasion these differences are important.

In both Gosu and Java, the language primitive types like `int` and `boolean` work differently from objects (descendents of the root `Object` class). For example:

- you can add objects to a collection, but not primitives
- variables typed to an object type can have the value `null`, but this is not true for primitives

The Java classes `java.lang.Boolean` and `java.lang.Integer` are `Object` types and can freely be used within Gosu code because of Gosu's special relationship to the Java language. These wrapper objects are referred to as *boxed types* as opposed to the primitive values as *unboxed types*.

Gosu can automatically convert values from unboxed to Java-based boxed types as required by the specific API or return value, a feature that is called *autoboxing*. Similarly, Gosu can automatically convert values from boxed to boxed types, a feature that is called *unboxing*.

In most cases, you do not need to worry about differences between boxed and unboxed types because Gosu automatically converts values as required. For example, Gosu implicitly converts between the native language primitive type called `boolean` and the Java-based object class `Boolean` (`java.util.Boolean`). In cases you want explicit coercion, simply use the "`as ...NEWTYPE`" syntax, such as "`myIntValue as Integer`".

If your code implicitly converts a variable's value from a boxed type to a unboxed type, if the value is `null`, Gosu standard value type conversion rules apply. For example:

```
var bBoxed : Boolean
var bUnboxed : boolean

bBoxed = null      // bBoxed can genuinely be NULL
bUnboxed = bBoxed  // bUnboxed can't be null, so is converted to FALSE!
```

For more information, see "Type Object Properties" on page 306.

Deploying Your Java Classes

The only strict requirement for creating Gosu-to-Java classes is to put the compiled classes in special directories. However, in practice, most Gosu-to-Java classes use the entity libraries, so it is critical to also correctly build and compile against the right version of the entity libraries.

Anytime you change the data model due to extensions or customizations, you must regenerate the Java API libraries so that you can compile your Java code against it.

IMPORTANT After you change the data model, regenerate the Java API libraries, which includes the entity libraries. For more information about entity libraries, see “Java Entity Libraries Overview” on page 114. For instructions on regenerating the libraries, see “Regenerating the Integration Libraries” on page 17 in the *Integration Guide*.

Detailed Java Class Deployment Checklist (As Command Line Tool)

If you are using Gosu as a self-contained command-line tool, you can deploy Java classes that your Gosu can call. You can put your class files and JAR files wherever you want. Your Java program can contain a directive that identifies one or more class path directories to look for class files and JAR files. The classpath can be a relative path or an absolute path. For details, see “Gosu Program Structure” on page 321.

Detailed Java Class Deployment Checklist (In ClaimCenter)

The following steps describe installation for a typical deployment of a Java class accessible from Gosu:

IMPORTANT This deployment checklist only applies to Java classes called directly from Gosu but not for Java plugins (Java classes that implement a ClaimCenter plugin interface). If you are writing a Java plugin, see “Deploying Java Plugins” on page 115 in the *Integration Guide*.

1. Determine which Java classes you need to deploy.
2. Regenerate the Java API libraries. If you did not update the data model since the last time you regenerated the Java API libraries, skip this step. For details, see “Regenerating the Integration Libraries” on page 17 in the *Integration Guide*.
3. In your Java code, if you use Guidewire entity types, remember to import and reference full entities (“external” Java classes) appropriately from the Java API libraries. For example, reference the ClaimCenter `Address` object as the class `com.guidewire.cc.external.Address`. For more information about this, see “Java Entity Libraries Overview”, on page 114.

   Note: Do not use the SOAP entities (for example, `com.guidewire.cc.webservices.entity.Address`) from Java code except in cases where you need to call SOAP APIs from within a Java class.
4. If you use Guidewire entity types, compile your Java class against the entity libraries at the path:
   `ClaimCenter/java-api/lib/gw-entity-cc.jar`
5. Copy your own compiled **Java class files** to:
   `ClaimCenter/modules/configuration/plugins/Gosu/classes`

   IMPORTANT You may need to create the `Gosu` directory if it is not already present. Carefully note the capitalization of `Gosu`, as it does not work as `gosu`. Note also that JAR files must not go in this directory. See the following item for more information.

   If the class files are not found in that directory, Gosu looks for the classes in the shared directory:
   `ClaimCenter/modules/configuration/plugins/shared/classes`
6. Copy necessary Java library **JAR files** or that your code needs into the directory:
   `ClaimCenter/modules/configuration/plugins/Gosu/lib`

   IMPORTANT You may need to create the `Gosu` directory if it is not already present. Carefully note the capitalization of `Gosu`, as it does not work as `gosu`. You can alternatively replace `Gosu` in that path with the string `shared`, to store the files in the shared library directory.

If the libraries files are not found in that directory, Gosu looks for them in:

`ClaimCenter/modules/configuration/plugins/shared/lib`

7. Rebuild the application WAR file using the `build-war.bat` script to generate a new ClaimCenter web application file.
8. If Guidewire Studio is running, quit and restart Guidewire Studio so it can find the new libraries.

The following Gosu code example uses a custom Java class:

```
/// in an event business rule defined in Guidewire Studio...

var claim = messageContext.Root as Claim
var myGosuPlugin = new com.mycompany.myClass()

var myMessageText = myGosuPlugin.generateDynamicMessage(claim)

// generate a message with myMessageText...
```

Class Loading for Java Classes Called from Gosu

Java classes called from Gosu use a Java class loader that loads Java classes in the following order:

1. Attempt to load from `ClaimCenter/modules/configuration/plugins/Gosu/classes`
2. If not found yet, load from `ClaimCenter/modules/configuration/plugins/Gosu/lib`
3. If not found yet, load from `ClaimCenter/modules/configuration/plugins/shared/classes`
4. If not found yet, load from `ClaimCenter/modules/configuration/plugins/shared/lib`
5. If not found, the class *delegate loads.*

For more information about delegate loading, see “Delegate Loading” on page 110.

For more information about Java class loading in general, see “Java Class Loading, Delegation, and Package Naming” on page 108.

Java Class Loading, Delegation, and Package Naming

Java Class Loading Rules

If loading custom Java code into Gosu or if accessing Java classes from Java code, the Java virtual machine must locate the class file with a *class loader.* Class loaders use the fully-qualified package name of the Java class to determine how to access the class.

ClaimCenter follows the rules in the following list to load Java classes, choosing the first rule that matches and then skipping the rules listed after it:

1. **General delegation classes.**

   The following classes *delegate load*:

   - `javax.*` - Java extension classes
   - `org.xml.sax.*` - SAX 1 & 2 classes
   - `org.w3c.dom.*` - DOM 1 & 2 classes
   - `org.apache.xerces.*` - Xerces 1 & 2 classes
   - `org.apache.xalan.*` - Xalan classes
   - `org.apache.commons.logging.*` - Logging classes used by WebSphere

   For more information about delegate loading see, “Delegate Loading” on page 110.

2. **Guidewire full Java entities, plugin classes, and utility classes.**

Classes in the following JAR files *delegate load*:

- `gw-plugin.jar`
- `gw-plugin-cc.jar`
- `gw-entity-cc.jar`
- `gw-util.jar`, except for classes in `com.guidewire.util.webservices.*`

For more information about delegate loading see, "Delegate Loading" on page 110.

3. **Guidewire SOAP entities, plugin classes, and utility classes.**

   Classes in the following JAR files *locally load*:

   - `gw-soap-cc.jar`
   - `gw-util.jar`, including all the classes in `com.guidewire.util.webservices.*`

   For more information about local loading see, "Local Loading of Your Classes" on page 109.

4. **All your classes.**

   If the package does **not** begin with `com.guidewire.*`, then *load locally.*

   WARNING Java classes you deploy must **never** have a fully-qualified package name that starts with `"com.guidewire."` because that interferes with class loading behavior.

5. **Internal classes.**

   If the class is an internal class, then the class *delegate loads*.

   WARNING Java code you deploy must **never** access any internal classes other than supported classes and documented APIs. Using internal classes is dangerous and unsupported. If in doubt about whether a class is supported, immediately ask Customer Support.

   If you need to access entities and supported domain methods on those entities, use the generated *entity libraries*. For more information, see "Java Entity Libraries Overview" on page 114.

Local Loading of Your Classes

For Java plugins

ClaimCenter Java plugin class loader follows these rules for loading a class locally from the deployed web application in *SERVER*`/webapps/cc/....`

1. The Java plugin class loader looks for the class in the plugin directory you define in the plugin registry in Studio. Given the plugin directory name *`plugindir`*, the server looks in the directories:
   - *SERVER*`/webapps/cc/modules/configuration/plugins/`*plugindir*`/classes`
   - *SERVER*`/webapps/cc/modules/configuration/plugins/`*plugindir*`/lib`
2. If the class was not found in the plugin's local repository, or the plugin did not define a `plugindir` repository, the class loader looks in the shared repository directories:
   - *SERVER*`/webapps/cc/modules/configuration/plugins/shared/classes`
   - *SERVER*`/webapps/cc/modules/configuration/plugins/shared/lib`
3. If the class was not found in the shared repository, then the class *delegate loads*.

Be sure to read the following topics if you are deploying a Java plugin:

- "Plugin Overview" on page 101 in the *Integration Guide*

- “Deploying Java Plugins” on page 115 in the *Integration Guide*

IMPORTANT If you are writing a Java plugin, read “Plugin Overview” on page 101 in the *Integration Guide* and “Deploying Java Plugins” on page 115 in the *Integration Guide*.

Class Loading for Java Classes Called from Gosu (not Java plugins)

Java classes called from Gosu also use a plugin class loader to load Java classes. The name of its local repository within the plugins directory is `Gosu`.

For example, Gosu loads Java classes in the following order:

1. Attempt to load from `ClaimCenter/modules/configuration/plugins/Gosu/classes`
2. If not found yet, load from `ClaimCenter/modules/configuration/plugins/Gosu/lib`
3. If not found yet, load from `ClaimCenter/modules/configuration/plugins/shared/classes`
4. If not found yet, load from `ClaimCenter/modules/configuration/plugins/shared/lib`
5. If the class is in the `shared/basic` folder, ClaimCenter loads the class and disables the special collection conversion behavior. In other words, these two directories suppress the conversion behavior:
   - *SERVER*`/webapps/cc/modules/configuration/plugins/`**shared/basic**`/classes`
   - *SERVER*`/webapps/cc/modules/configuration/plugins/`**shared/basic**`/lib`

   IMPORTANT The collection conversion suppression feature is not available for all situations. For important details about this API, see “Disable Container Conversion In Some Cases” on page 111.

6. If not found, the class *delegate loads.*

See “Java Class Repository Listing” on page 111 for a full list of Java class repositories.

For more information about calling Java classes from Gosu, see “Overview of Calling Java from Gosu” on page 102.

Delegate Loading

If the ClaimCenter plugin class loader *delegates* Java class loading, ClaimCenter requests the application server class loader to load the class. Each application server (such as Tomcat, WebSphere, Weblogic) has its own class loader rules. In general, an application server tries to load classes in the following order:

1. Attempt to load the classes from the application’s local repository directories:
   - *SERVER*`/webapps/cc/WEB-INF/classes`
   - *SERVER*`/webapps/cc/WEB-INF/lib`

   However, typically there are classes that the application server class loader does **not** attempt to load from the application’s local repository and instead skip to the next step. Most containers, including Tomcat, do not load Java extension classes or other classes listed as “General delegation classes” in “Java Class Loading Rules” on page 108 in this way.
2. If not found yet, load from bootstrap classes of the Java virtual machine.
3. If not found yet, load from system class loader classes
4. If not found yet, load from the shared *application server*’s repositories, the exact locations of which can vary by web application server.

For Detailed Application-Server-specific Behavior

For more details about your web application server’s class loading behavior, refer to the following web pages:

- For WebSphere version 6, and also version 5 since it has similar behavior:

```
http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/topic/com.ibm.iea.was_v6/was/6.0/
  Runtime/WASv6_ClassLoader_Overview.pdf
```

- For Tomcat 6:

```
http://tomcat.apache.org/tomcat-6.0-doc/class-loader-howto.html
```

Java Class Repository Listing

Java classes loads from various locations depending on the fully-qualified package symbol.

The following files within *SERVER*/webapps/cc/modules/configuration/config/... exactly mirror the structure of files within your configuration environment ClaimCenter/modules/configuration/config/....

```
SERVER/webapps/cc/config                             Root of your configuration dir
SERVER/webapps/cc/modules/configuration/plugins
SERVER/webapps/cc/modules/configuration/plugins/Gosu           Java classes from Gosu
SERVER/webapps/cc/modules/configuration/plugins/Gosu/lib       Java classes from Gosu libs
SERVER/webapps/cc/modules/configuration/plugins/Gosu/classes   Java classes from Gosu classes
SERVER/webapps/cc/modules/configuration/plugins/shared
SERVER/webapps/cc/modules/configuration/plugins/shared/classes    Local load - shared directory classes
SERVER/webapps/cc/modules/configuration/plugins/shared/lib        Local load - shared directory libs
SERVER/webapps/cc/modules/configuration/plugins/plugindir
SERVER/webapps/cc/modules/configuration/plugins/plugindir/classes Local load - explicit dir classes
SERVER/webapps/cc/modules/configuration/plugins/plugindir/lib     Local load - explicit dir libs
```

The following lines indicate directories within the web application, and their role in Java class loading:

```
SERVER/<web app container shared+internal JARs>     Delegate load - shared web app container repository
SERVER/webapps/cc                                    Root of ClaimCenter web application
SERVER/webapps/cc/WEB-INF
SERVER/webapps/cc/WEB-INF/classes                    Delegate load - app local repository classes
SERVER/webapps/cc/WEB-INF/lib                        Delegate load - app local repository libs
```

For more information about delegate loading, see “Delegate Loading” on page 110. For more information about local loading, see “Local Loading of Your Classes” on page 109.

Special Handling of Collections, Arrays, Maps

Gosu specially handles containers (arrays, collections, and maps) when calling out to Java, by default. First, Gosu makes a new copy of the container. Next, Gosu converts each element in the original container and adds it to the new container. Effectively, the new container is a clone of the original container, and whose elements are converted versions of the elements in the original.

For arrays, the libraries create arrays with the appropriate class, including any appropriate conversions if the array is an array of Guidewire entity instances. For instance, an array of `Address` objects convert from the internal (Gosu) `Address` class to the external (Java) `Address` class. For more information about working with entities from Java, see “Java Entity Libraries Overview” on page 114.

Disable Container Conversion In Some Cases

If you are writing Java classes that implement ClaimCenter plugin interfaces (also called *Java plugins*), you always must be aware of this behavior. This affects any collections that are argument types or return types for object methods for a Java class.

If you are writing other types of Java code, you can put your Java classes and libraries in a special place to disble this behavior. However, this only works if **all of the following are true**:

- none of your class’s methods have argument types or return types that are ClaimCenter entity instances
- none of your class’s methods have argument types or return types that are containers of ClaimCenter entity instances. For example, no arrays of `Address`, or `ArrayList<Person>`.
- any Java classes are not registered as a Java plugin in Studio. In other words, your Java class does not implement a ClaimCenter plugin interface.

To disable container conversion for your Java classes or libraries

1. Create the following two directories:

   `ClaimCenter/modules/configuration/plugins/shared/basic/classes`
   `ClaimCenter/modules/configuration/plugins/shared/basic/lib`

2. Put any of your `classes` (`.class` files) in the `classes` directory (using standard Java hierarchies for the package name).

3. Put any of your libraries (`.jar` files) in the `lib` directory.

Generally speaking, put any general purpose third-party JAR files in this special libraries directory so that no that conversion happens for those classes.

Details of Container Conversion

External Java code **cannot modify the Gosu version of the container directly** (also called *in-place modification*). Because the Java code is operating with the cloned copy of the list, direct modifications in Java are not visible to the Gosu caller.

If any of the elements of the container is another container, that container is also converted. This continues recursively until all containers are converted.

When converting entity instances, ClaimCenter converts it to the appropriate internal (Gosu) or external (Java) interface. For more information about working with entities from Java, see "Java Entity Libraries Overview" on page 114.

Follow the following guidelines for passing containers from Gosu to Java:

- Never pass a container between Gosu and Java and expect the other side to modify the container such that the container changes are visible after the function call. That never works because the containers reference different objects on the two sides of the function call.

  IMPORTANT Do not pass a container between Gosu and Java and expect the other side to modify the container such that the container changes are visible after the function call.

- Avoid passing large generic container objects (such as a large `List`) between Gosu and Java. Conversion of large containers is very resource-intensive.
- Remember that this conversion only happens for method calls, not for accessing object instance variables.
- Be careful with your assumptions about object reference equality. After you pass a container from Gosu to Java, or from Java to Gosu, they are different objects. If you pass a container from Gosu to Java, and it returns a container again, the return result is a different object from your original container. Remember that if you use the Gosu equality operators (both == and ===), they return `false` when comparing different container references.

  IMPORTANT Be careful with your assumptions about object reference equality after passing containers between Gosu and Java.

The following table summaries the implications of conversion for different types of objects.

Container type	How ClaimCenter Converts these container
arrays	If arrays move across the entity library interface, the libraries create a new array with a type that matches the type of the existing array. For example, an array of internal `User` objects converts to an array of external `User` objects. If the internal array is untyped, the converted array uses the argument type or return type that the caller expects.

Container type	How ClaimCenter Converts these container
`java.util.Collection` (this includes `java.util.ArrayList`	If a `Collection` passes across the entity library interface, the entity library interface creates a new instance of the same class used for the collection. This might not be possible in some cases. For example, it might fail if a `List` created using `Collection.singletonList()` has an implementing class with no public *no-argument constructor*. In this case, ClaimCenter substitutes the class `ArrayList` for `List` objects and `HashSet` for `Set` objects.
`java.util.HashMap`	The same issue mentioned for `java.util.Collection` also applies to `Map` objects. The entity library interface attempts to create a new instance of the current class, and if that is not possible, the libraries use a `HashMap`.

Objects other than those mentioned in this table have no conversion. This is why the strategy referred to earlier with wrapping a collection in a custom class works properly. This means, however, that you must use only method calls to add or remove elements rather than directly access public member variables on those objects. Conversion only happens on method calls. Conversion does not happen for member variable access.

The libraries maintain relations between *references* within the converted arguments even as those objects themselves are converted. For example, suppose you write a Java class accessible to Gosu. That class declares a method `doSomething(Activity a1, Activity a2)` invoked from Gosu as `doSomething(a, a)` such that the same entity might be passed for both arguments. If the both parameters are the same entity, from the external side of the entity libraries in Java, the statement `a1 == a2` evaluates to `true`. This evaluates to `true` because the graph of references within the method call remains even after argument conversion.

This also applies to objects contained within a `Map`, `Collection`, or arrays, including the case where those containers contain themselves. For example, suppose a map contains the same object within the mappings `Field1` and `Field2`. In other words, suppose `mymap.Field1` references the same object as `mymap.Field2`. If a map passes across the entity library interface, the entity libraries covert the objects passed in the `Map`. Because they were part of the same collection, the value of the `Field1` mapping is the **same** object as the value of the `Field2` mapping. This is true even though the object itself in that property converts and is different from the value that `mymap` had for that property before the conversion.

Example: Working with Collections Safely

To pass collections as method parameters or return values without duplicating the container at all, wrap the container in a custom Java class (or Gosu class). To manipulate the data, ensure this outer class has methods that can manipulate the collection inside it. For example, add methods that **add or remove items** from the collection. If the outer class has a method that returns a container, Gosu converts the return value and returns the clone of the collection.

For example, suppose you want to call a Java method from Gosu with a Java `Map`. Suppose you want the `Map` to load in your Gosu code and then call out to Java to modify the `Map`. Create a *wrapper class* with an inner collection in a private variable. Add a method that adds an item to the collection. This way, the parameters convert but not the entire collection. Because Java maps are native Java objects, the Java code can modify them directly. The entity libraries do not convert the entire collection for every method call.

For example, suppose you want to define a class representing an English sentence. A sentence consists of a list of words. One way to model this would be the following example:

```
public class Sentence {
  private ArrayList<String> words;

  public Sentence() {
    words = new ArrayList<String>();
  }

  public List<String> getWords() {
    return words;
  }
}
```

The problem with this class is that there is no way to add words to the sentence from Gosu.

It might seem like you could try the following Gosu code:

```
// bad example
var sentence = new Sentence()
sentence.Words.add("Once")
```

The Java method `getWords` becomes a property access in Gosu (the `Words` property) because it starts with `"get"`. Even though Gosu considers this a property accessor, it is really a method call, so ClaimCenter creates a **copy** of the `Words` list before returning it to the Gosu caller. The last line of that example adds the list element `"Once"` to the copy of the list. However, this is not the actual list maintained inside the `Sentence` object.

The recommended approach is to add the mutator methods (for example, add or remove) to the `Sentence` class:

```
public class Sentence {
  private ArrayList<String> words;

  public Sentence() {
    words = new ArrayList<String>();
  }

  public void addWord(String word) {
    words.add(word);
  }

  public void insertWord(int position, String word) {
    words.add(position, word);
  }

  public List<String> getWords() {
  return words;
  }
}
```

To write Gosu code that adds words, use something like the following:

```
var sentence = new Sentence()
sentence.addWord("Once")
```

Java Entity Libraries Overview

An *entity* is a ClaimCenter object defined in the data model with built-in properties and optional data model extension properties. For example, `Claim` and `Address` are examples of entities. Write Java code that accesses *entities* from Java plugin code and from Java classes called from Gosu. If you regenerate the Java API libraries, ClaimCenter generates Java libraries that include what Guidewire calls the *entity libraries*. To regenerate these files, see “Regenerating the Integration Libraries” on page 17.

You can copy the entity libraries to special locations and then compile Java code against these libraries. Your Java code can get or set entity data or to call any exposed additional methods (called *domain methods*) similar functionality as from Gosu. For example, a `Message` object as viewed through the entity library interface has data getter and setter methods to get and set data. For example, `getPayload` and `setPayload`. Additionally, the object has additional methods that trigger complex logic, such as `Message.submitAck(...)`.

Understanding and using the entity libraries is critical in the following situations:

- **Java plugin development.** To implement most plugin interfaces in Java, you must use the entity libraries. Typically, plugins define parameters or return values that include entities or collections, and the libraries convert such data. Also, most plugins require access entity data or must call entity domain methods to perform their required action.
- **Java classes called from Gosu.** If your Java code uses entity types or you specially exported some Gosu classes, you must use entity libraries. The libraries convert data between Gosu and Java objects if data types are not native to Java, as well as special handling of collections. Your code can compile against these libraries and access properties and methods of Guidewire entities and Gosu classes. Once you deploy your Java class files in the right location, any Gosu code can access these Java classes.

For more information about what circumstances you use the Java entity libraries, and the differences between the full Java entity libraries and the *SOAP entity* libraries, see:

- “Required Generated Files for Integration” on page 20
- “Diagram of Entity Library Accesses and Differences” on page 22.

The generated entity library interfaces have a defined naming convention and only expose the appropriate public methods of each object. ClaimCenter packages generated interfaces and classes in a Java archive (JAR) file for you to use.

Regenerating these files generates a library JAR file containing interfaces for each ClaimCenter entity, and defines them within the `com.guidewire.cc.external.entity` package. To regenerate these files, see “Regenerating the Integration Libraries” on page 17.

Each interface extends from one of two different classes, depending on whether it is *keyable*. A keyable object is an entity in the Data Dictionary that contains the `keyable` attribute. If the entity is a subtype of another entity, the entity’s supertype defines whether the entity is keyable.

- If the class is keyable, it extends from `com.guidewire.external.entity.KeyableEntity`
- If the class is not keyable, it extends from `com.guidewire.external.entity.Entity`

For example, reference the `Address` as `com.guidewire.cc.external.Address`.

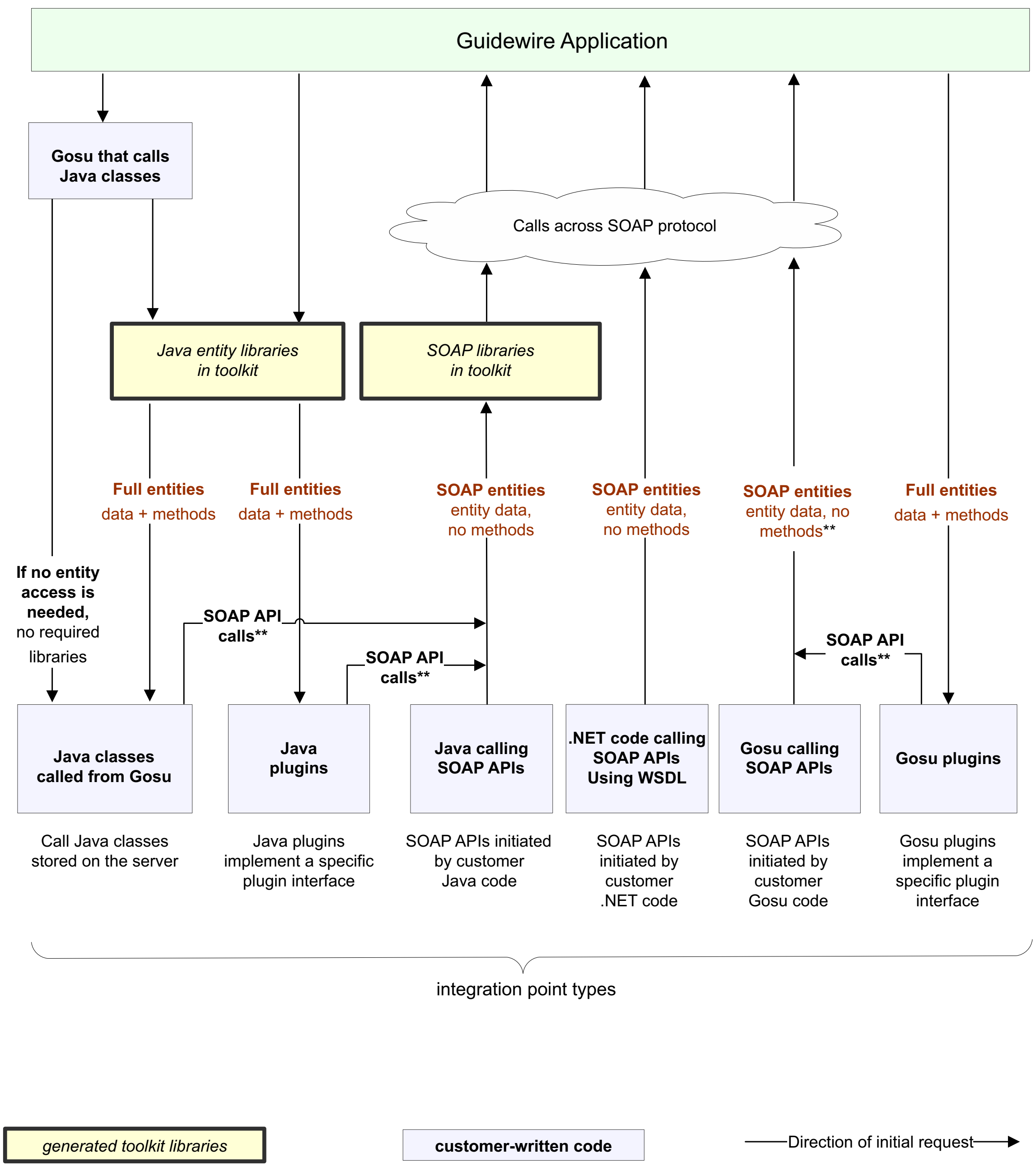
Almost every entity property or domain method available from Gosu appears in the generated class, with a few exceptions, as explained later in this section. This includes all generated properties, all virtual methods. However, there are some methods that appear only in certain contexts, such as within business rules or within Gosu templates, and not all of these appear on external interfaces.

Diagram of Entity Library Accesses and Differences

The following diagram illustrates the entity access differences between integration points. For differences between integration points and the files required for each, see “Integration Overview”, on page 13 and “Required Generated Files for Integration” on page 20.

WARNING Avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine in production systems. Be careful about any SOAP calls to the same server. If the SOAP API hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data. Instead, refactor your code to avoid this case. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance. If you have questions about how to convert some particular use locally-hosted SOAP APIs from plugins or rules, contact Customer Support. This is true for all types of local loopback SOAP calls to the same server. This includes soap.local.* objects, the SOAP client Java classes from Java plugins, and Studio-registered web services that call the same server as the client.

Guidewire Entity Access



** Always avoid SOAP API calls that loop back to the **same** server from any plugin code or rule set execution, particularly if the API changes entity data. Contact Customer Support if you think you need to do this. Also note that calls to SOAP APIs always use SOAP entities, which have entity data but no domain methods. In cases where full entities are otherwise available, SOAP entities exist in a totally separate package hierarchy from regular "full" entities.

Using Java Entity Libraries

Anytime you change the data model due to extensions or customizations, you must regenerate the entity libraries so that you can compile your Java code against it.

To regenerate entity libraries for use in integration, regenerate (rebuild) the Java API libraries. See “Regenerating the Integration Libraries” on page 17 for details.

ClaimCenter creates the Java API library JAR files in the following directory:

```
ClaimCenter/java-api/lib/...
```

This directory includes several JAR files, include the Java entity libraries in the file `gw-entity-cc.jar`.

You must **copy** the `gw-entity-cc.jar` file to the following directory for ClaimCenter to find it at run time:

```
ClaimCenter/modules/configuration/plugins/shared/lib
```

IMPORTANT Either manually copy the `gw-entity-cc.jar` file to your `configuration` module after regenerating the file, or modify your build system to copy this file for you.

Compile your code against those entity library files. The JAR file contains compiled classes for each of the Guidewire entities and typelist classes, and the classes `Entity`, `KeyableEntity`, `TypeKey`, `Key`, and `GenericCheckedException`.

Refer to the documentation for the generated entity libraries at the path:

```
ClaimCenter/java-api/doc/api/index.html
```

IMPORTANT The generated library files in `ClaimCenter/java-api/lib/*.jar` are the only supported versions of the Guidewire Java libraries. Do not use any libraries exist in internal modules in the product hierarchy.

Related Deployment Information

- If you implement a Java plugin, see “Plugin Overview”, on page 101 and “Deploying Java Plugins” on page 115.

Creating Entities from Java

Java code sometimes needs to create new entities. For example, document management integration code creates a `Document` entity to return to ClaimCenter. For more information about creating new entities from Java code using the `EntityFactory` class, see “Java Entity Utility APIs” on page 121.

Getting and Setting Properties from Java

Entity properties (as well as any object properties from Gosu if accessed from Java) are on the external interfaces as getter and setter methods. Getter and setter methods are methods to get or set properties and their names start with the text “`get`” and “`set`”. A readable and writable property named *`MyField`* shows up in the interface as a pair of `get`*`MyField`* and `set`*`MyField`* methods. Read-only properties do not expose a `set` method on the object. If the property named *`MyField`* contains a Boolean value, it appears in the interface slightly differently as `is`*`MyField`* instead of `get`*`MyField`*.

Examples:

```
address.setFirstName("Russ");
lastName = address.getLastName();
tested = someEntity.isFieldname();
```

Extension properties appear in the generated classes in the exact same way as built-in properties. ClaimCenter adds associative properties to an entity automatically based on typelist values.

Calling Entity Domain Methods from Java

Almost all ClaimCenter object methods visible to Gosu appear on the generated Java entity interface. Most entity methods on entity instances appear as regular methods, for example:

```
claim.addEvent("CustomEventName");
```

A few methods do not export to Java due to differences in certain internal classes that cannot convert to the external Java interface.

Exception Handling from Java

Guidewire specifies that some entity methods throw exceptions. These exceptions are usually internal Guidewire classes. Exception classes are internal to ClaimCenter and unavailable within the external entity libraries. Because of this, the generated interface cannot directly declare the generated method as throwing exactly the same exceptions as the internal exceptions.

To workaround this problem, entity interfaces throw the exception `com.guidewire.external.GenericCheckedException` if the underlying method can throw a checked exception. The entity interfaces throw `java.lang.RuntimeException` if the class explicitly declares the throwing of some variant of the `RuntimeException` exception. Find Guidewire exceptions declared on the underlying method for each entity in the Javadoc documentation for that method within the API Reference Javadoc documentation. If an exception occurs while calling the method, Gosu rethrows the original runtime exceptions to the calling code. ClaimCenter wraps any checked exceptions in a `GenericCheckedException` and then throws the exception.

TypeKey Classes from Java

The entity libraries include a class for each typelist generated into the `com.guidewire.cc.external.typelist` package and implementing the `com.guidewire.external.typelist.TypeKey` interface.

Each class contains a set of static instances of the class, one per non-retired `typekey` in the `typelist`. New instances of the class cannot be created directly, ensuring that the `typekey` constants form a type safe enumeration, but they can be reflectively looked up using the method:

```
entity.getByCode(String code);
```

All *typekey* constants use the typekey's code in all upper case as their name and without any "`TC_`" prefix. Java identifiers only allow letters, numbers, and the underscore character, so any other characters convert to underscores in the constant name. In addition, identifiers cannot start with numbers, so any typecode that starts with a number results in a constant whose name starts with an underscore.

For example, the `CheckType` typecode `Primary` would be referenced within the generated libraries as `com.guidewire.cc.external.typelist.CheckType.PRIMARY`.

Parameterization of Types Stripped from Entities in Java

If any plugin definition uses parameterized types, ClaimCenter strips parameterization from those types when generating external entities in the Java entity libraries. This applies to both method parameters and return types

For example, if a plugin interface returns the type `List<User>`, in Java it simply returns the type `List`.

Special Entity-related Java Classes

There are five entity-related classes that represent Guidewire internal classes, which at generation time copies into the generated entity library packages:

Root of Entity Hierarchy

The root of the entity hierarchy and the superclass for any non-keyable entities is the class:

```
com.guidewire.external.entity.Entity
```

Superclass for Keyable Entities

The superclass for all keyable entities is the class:

```
com.guidewire.external.entity.KeyableEntity
```

This extends from `com.guidewire.external.entity.Entity`.

Entity's Key External Representation

The external representation of an entity's key is the class:

```
com.guidewire.external.entity.Key
```

Superclass for Generate Typelist Classes

The superclass for each generated typelist class is the class:

```
com.guidewire.external.typelist.TypeKey
```

These have the name `TypeKey` because a `typelist` is really a set of `TypeKey` instances.

Generic Checked Exceptions

Exceptions declared and thrown if a method can throw checked exceptions derive from the class:

```
com.guidewire.external.GenericCheckedException
```

Argument Conversion in Java Entity Libraries

If you pass arguments back and forth between Java and either Gosu or internal ClaimCenter code, arguments must convert between the internal format and the external Java format. Because the process is complex, you must understand the conversions or you experience unexpected behavior.

The most important things to remember to reduce the chance of unexpected behavior are as follows:

- Always assume that references to entities and keys can change across calls. For more information, see "How Entities and Typekeys Convert to and from Java" on page 120.
- Remember that Gosu copies and converts `Map`, `Array`, and `Collection` objects in both directions. Because of this, these objects cannot be effectively modified on the other side of the entity library interface. They are often memory-intensive and resource-intensive to pass across.

IMPORTANT It is extremely critical to understand how ClaimCenter handles collections, arrays, and maps. This conversion behavior can be suppressed in some cases. For more information, see "Special Handling of Collections, Arrays, Maps" on page 111.

A call from Gosu or ClaimCenter application logic out to external Java code must convert *internal objects* (Gosu objects) to *external objects* (objects viewed through the external interfaces). Similarly, if a Java method returns a value, the return value automatically converts from external to internal formats.

Note: For purposes of this section, *external* classes refer to Java entity classes and *internal* classes are classes used by Gosu code and internal application logic.

A call from a Java class to an internal domain method must convert external objects to internal objects. The libraries convert internal objects into external objects as the method returns.

How Entities and Typekeys Convert to and from Java

Comparing Entities

If a ClaimCenter entity converts from internal form to external form, a dynamic proxy wraps the internal entity and implements the external interface. If the entity converts back to internal objects, that proxy returns the original (internal) entity. This preserves references from the internal side (internal code and Gosu). If an `Address` passes to plugin code and the plugin returns it, the return value is the same object passed to the plugin from the internal code or Gosu world. However, this is not true from the Java side of the entity libraries. Since the libraries create a new proxy every time for Java code using these libraries, you **cannot** rely on the == operator to determine equality.

For example, this expression always evaluates to `false`:

```
activity.getClaim() == activity.getClaim()
```

However, the methods `equals`, `toString`, and `hashCode` call through to the wrapped entity and they are safe to use. Thus, in contrast to the earlier example, the following expression evaluates to `true`:

```
activity.getClaim().equals(activity.getClaim())
```

And this expression also evaluates to `true`:

```
activity.getClaim().hashCode() == activity.getClaim().hashCode()
```

Comparing Typekeys

As typekeys convert from internal objects to external objects, the external interface provides the appropriate external `Typekey` constant. Or you can use the `getOrCreateByCode` method on the external `Typekey` class to get a typekey. As typekeys convert from external to internal format, Gosu retrieves the internal typekey based on the code.

This means that if you create an external `typekey` that does **not** actually exist in the system, the conversion attempt fails. Reasons that the external typekey would not exist include getting results from plugin code to `getOrCreateByCode` or if you remove type keys without regenerating external entity classes.

Since typekey instances are currently *singletons* (a single instance of the class) both internally and externally, `typekey` references preserved across calls unlike entity references. However, this behavior **will change** in a future release and you cannot rely on the current behavior. To prepare for this future change, your Gosu code must now use `equals` instead of == for *typekey instances* as well as entities. Start to convert any code that relies on == for typekeys translated through the entity libraries, and write new ClaimCenter code to include this requirement.

For example, the following expressions return `true` in the current release but returns `false` in a future release:

```
claim.getState() == claim.getState()
claim.getState() == ClaimState.OPEN
```

Instead, change similar expressions to something like:

```
claim.getState().equals(ClaimState.OPEN);
```

IMPORTANT In a future release, the behavior of the == operator will change for typekeys accessed through the entity libraries from Java code. Similarly, the behavior will change for typekeys passed back to Gosu from Java classes. Begin converting any existing code now, and write any new code to satisfy this requirement.

Comparing Keys

The entity libraries also wrap key objects (the Key type) when converting from internal to external format. The libraries unwrap the object going from external to internal format.

As with entities, from the Java side, you **cannot** assume references preserve between calls, so use `equals()` instead of the equality operator == for equality comparisons.

Java Entity Utility APIs

Creating and Accessing Entities From Java

To create an entity, first get the current *entity factory* instance, which is the object capable of creating new entities from Java code. Create this instance by calling the `com.guidewire.external.entity.EntityFactory` class's static method `getInstance`, which returns an `EntityFactory` instance. The entity factory instance contains important methods such as `newEntity(Class)`.

First get a reference to an entity factory instance, as described earlier in this section. Next, call the entity factory's method `newEntity(Class)` method.

For example, to get new entities from Java:

```
import com.guidewire.external.entity.EntityFactory;
...
a = EntityFactory.getInstance().newEntity(Address.class)
```

Any references to entities modified or returned from a plugin method commit to the database after the method returns, unless otherwise specified in the API Reference Javadoc documentation. In all cases, after the method returns you cannot rely on the values of that object after the method return. Also, you cannot assume that changes to the object after that point remain.

If you need to make changes later to the object, read and save the *public ID* of the object. If necessary, your plugin (or an external system that interfaces with your plugin) can use the ClaimCenter web services API to read or modify the object as necessary.

There are important differences in the way you create SOAP entities versus full Java entities. To create a SOAP data-only entity from Java, use the `new` keyword:

```
import com.guidewire.cc.webservices.entity.Address;
...
a = new Address();
```

In contrast, create a full entity using the entity libraries in Java with the `EntityFactory`:

```
import com.guidewire.external.entity.EntityFactory;
...
a = EntityFactory.getInstance().newEntity(Address.class)
```

If you must make changes later to the entity, read and save its key properties, such as the *public ID* of the object. If desired, use other APIs to find or modify the object from other code. For example, the `getEntityByRef` method discussed in "Finding Entities by Public ID Within Java Code" on page 122.

No Spawned Threads Except for MessageReply Callbacks

You can use `EntityFactory` class in the Java thread environments ClaimCenter sets up for you in the following contexts:

- **Java plugins in the plugin's initial thread context.** For more information. see "Plugin Overview", on page 101
- **Java code called by Gosu in its initial thread context.**
- **Java MessageReply plugin callback handler.** For more information, see "Implementing a Message Reply Plugin" on page 180.

With the exception of the message reply callback handler, spawned threads do **not** work with the `EntityFactory` class.

Mock Entity Customization for All Entities Created By the Entity Factory

For advanced testing of code like this, you can configure the entity factory to return *mock entities*. Mock entities, are special fake versions of entities to help you test code that uses entities. For more information, see "Testing Your Entity Code in Java (EntityMock)" on page 125 and "Modifying EntityFactory for Mock Entities" on page 127.

Finding Entities by Public ID Within Java Code

Using the Java entity libraries, if you populate a complex graph of data with references to entities, you can search for an entity by its public ID. For example:

```
myClaim = EntityFactory.getInstance().getEntityByRef(Claim.class, "cc:1234")
```

Which Java Types Export to Java Entity Libraries?

The following Java types accessed in Gosu export to the Java entity libraries in the following cases with the following differences:

Type of class	Exposed	Translated with intermediate proxy class
Built-in Java classes related to plugin interface method parameters or return types, or other critical integration tasks	Yes	Yes
Java classes in the `java.*` namespace	Yes	No
Java classes in the `javax.*` namespace	Yes	No
Additional classes exposed in the libraries: • `gw-plugin.jar` • `gw-plugin-cc.jar`	Yes	No
Your arbitrary Java classes	No	No

Non-entity Types in the Java Entity Libraries

Adding Gosu Non-Entity Types to the Entity Libraries

The primary purpose of the entity libraries is to allow you to write Java code to access Guidewire business entity objects. Additionally, you can make Gosu classes export to Java through the entity libraries to write Java code that interacts with built-in utility classes and your custom Gosu classes.

By default, ClaimCenter only generates Java classes for types that are entities, typelists, plugin interfaces, or a type attached to the previously-mentioned objects through their properties or methods. So, suppose your Gosu class is not already referenced by an entity, typelist, or plugin. ClaimCenter would not generate an external Java class for it.

IMPORTANT ClaimCenter does not automatically generate Java versions of all Gosu classes. To make a Gosu class available from Java, follow the instructions in this section.

To expose arbitrary Gosu types, add a line for that type in the configuration file:

```
ClaimCenter/modules/configuration/config/plugin/ExternalTypes.lst
```

The format of the file is simply one fully-qualified type (with the package as it exists in the Gosu type system) on each line. For example:

```
gw.api.util.CurrencyUtil
gw.api.util.DateUtil
```

Add additional types, such as other `gw.api.*` utility classes at the end of this file.

This works with all types that can be generated for external entities:

- all Gosu types
- all Java types in the `gw.*` namespace.

IMPORTANT This topic describes how to add these types to the entity libraries. To remap what packages in which they appear, refer to “Remapping Package and Type Names in Java Entity Libraries” on page 124. To instantiate these types, see “Using Gosu Classes from Java Entity Libraries” on page 123. To use static members, see “Static Methods and Variables in Java Entity Libraries” on page 123.

Using Gosu Classes from Java Entity Libraries

Gosu types automatically become part of the entity libraries in some cases, or you can manually add these types. See “Which Java Types Export to Java Entity Libraries?” on page 122 and “Adding Gosu Non-Entity Types to the Entity Libraries” on page 122 for details. After you regenerate the libraries these Gosu classes are available from Java.

By default, Gosu classes export to Java in the package hierarchy `external.*` as a Java class matching the original class name. For example, a Gosu class called `MyUtils` exports as the class `external.MyUtils`. All methods and properties dynamically translate from Gosu to Java and back again as needed. If you want to change the package for the class, refer to later in this section for package mapping configuration.

To construct a new instance of a Gosu class, use the class’s static object in its property called `CONSTRUCT`. That object contains methods that mirror the actual Gosu constructors.

For example, to create an instance of a Gosu class `MyClass` from Java using an empty constructor, use the code:

```
MyClass myInstance = external.MyClass.CONSTRUCT.newInstance();
```

If there are other constructors (in other words, constructors that take arguments), use the alternate exposed versions of the `newInstance()` method that have the same types of arguments. For example, to create an instance of a Gosu class `MyClass` from Java using a constructor that takes a `String` argument, use the syntax:

```
MyClass myInstance = external.MyClass.CONSTRUCT.newInstance("My argument to the constructor");
```

If you want to access methods and variables defined directly on the Gosu class, rather than on an instance of it, those are called static methods and variables. To access them, see “Static Methods and Variables in Java Entity Libraries” on page 123.

To remap the package your classes appear in the entity libraries, see “Remapping Package and Type Names in Java Entity Libraries” on page 124.

Static Methods and Variables in Java Entity Libraries

The entity libraries expose static methods and static variables on classes differently than natively through Gosu. The interfaces encapsulate all static methods and static objects in a new anonymous class that you can reference through the `UTIL` property of each class.

For example, suppose a class had a static property like this:

```
String MYSTATICFIELD = "Hello";
```

The external interfaces expose it as an anonymous class in the `UTIL` property:

```
static class UTIL {
```

```
  ...
  public static String getMYSTATICFIELD() {
    ...
  }
}
```

This means that you might expect to access static variables from Java code like the following:

```
String myStaticValue = MyClass.MYSTATICFIELD;
```

However, you must replace it with code of the form:

```
String myStaticValue = MyClass.UTIL.getMYSTATICFIELD();
```

Read Gosu class static properties with code like:

```
String geocodeStatus = MyClass.UTIL.getMYSTATICPROPERTY();
```

Set Gosu class static properties with code like:

```
String geocodeStatus = MyClass.UTIL.setMYSTATICPROPERTY("new value here");
```

Gosu Enhancements in Java Entity Libraries

Gosu enhancements export to Java code through the entity libraries. See the “Enhancements” on page 209 in the *Gosu Reference Guide* for more information about Gosu enhancements.

Both properties and methods defined in enhancements export to Java. There is no special task necessary to use this feature. Enhanced properties and methods are simply available on Guidewire entities after you regenerate the entity libraries (part of the Java API libraries).

Remapping Package and Type Names in Java Entity Libraries

Be default, external entity classes have the namespace of:

```
com.guidewire.cc.external.entity
```

Typelists have the namespace of

```
com.guidewire.cc.external.typelist
```

In this release, these namespaces are configurable in a file in the configuration called `ExternalMapping.properties`.

```
.../modules/configuration/config/plugin/ExternalMappings.properties
```

This file maps internal namespaces to external namespaces. To ensure backwards compatibility, the included mapping for entities remains.

The format for this properties file for **namespaces** is as follows:

```
INTERNAL_GOSU_NAMESPACE. = EXTERNAL_GOSU_NAMESPACE
```

Note the period after the internal Gosu namespace identifier.

If you wish to map a **single type**, then simply list the fully qualified name before and after for that type. In other words, the syntax is:

```
INTERNAL_TYPE_FULLYQUALIFIEDNAME = EXTERNAL_TYPE_FULLYQUALIFIEDNAME
```

For example:

```
gw.plugin.addressbook.IAddressBookAdapter = com.guidewire.ab.plugin.addressbook.IAddressBookAdapter
gw.plugin.document.IDocumentContentSource = com.guidewire.ab.plugin.document.IDocumentContentSource
```

By default it contains one entry:

```
entity. = com.guidewire.cc.external.entity
```

IMPORTANT This file affects the package mapping, but does not affect whether a type exports to external libraries. For that, see “Adding Gosu Non-Entity Types to the Entity Libraries” on page 122.

Note that you can change the type name entirely in addition to an optional package change, for example:

```
example.MyInternalClass = example.external.MyExternalClass
```

Testing Your Entity Code in Java (EntityMock)

To help you test Java code that uses entity libraries, ClaimCenter includes a utility class called `EntityMock.` This class generates a *mock* (fake) instance of a ClaimCenter entity.

For example, suppose you were writing a `MessageTransport` plugin in Java to send messages to an external system. The main task of this plugin interface is to get the payload of a `Message` entity and send it using some sort of messaging system. Optionally, the plugin acknowledges that the message sent successfully. While developing the code, it might be useful to quickly test changes to the code without having to deploy it in a live ClaimCenter server as a ClaimCenter plugin.

For such cases, a Java unit test could use the `EntityMock` utility class to create a mock version of the `Message` entity. The unit test calls code that takes a `Message`, reads its properties, and sends the `Message`. After you test the code's logic, deploy and deploy and test your code in a real ClaimCenter servers.

In some cases, integration code that uses entities does more than just get and set properties. Fortunately, `EntityMock` allows you to fake the behavior of domain methods with as much (or as little) functionality necessary to test your code's behavior. Alternatively, extend the entity with new custom methods that might make testing easier.

How EntityMock Works

Creating a mock entity with `EntityMock` creates a proxy that implements setter methods (`set...`) on the entity to store properties of the entity and getter methods (`get...`) to return a stored values. By default, it also provides an implementation of `equals()`, `hashcode()` and `toString()` that works for most situations. After creating a mock entity, you may also add a *delegate object*. A delegate object is a class that you designate can handle (or pretend to handle) one or more methods on that type.

The delegate can override any method on the entity, as well as decorate the entity with additional methods for testing. If you add entirely new methods, you must extend the original entity interface to add signatures for the new methods, and then implement those methods on the delegate object.

If any code invokes a method on the mock entity, the proxy's invocation handler follows the following rules:

1. The handler examines the mock entity's delegate objects to see if any of them declare a method with exactly the same signature. If a matching method exists on a delegate object, the libraries invoke the method on the delegate.
2. If no such matching methods existed on delegate object, the handler checks if the method is `equals()`, `hashcode()`, or `toString()`. In any of those cases, the handler runs the default logic.
3. If neither of the rules mentioned earlier applies, the handler checks if the method is a getter or setter method and handles those by storing or retrieving values.
4. If any code calls other methods not explicitly defined by a delegate object, the handler throws an `IllegalArgumentException` exception.

Using EntityMock

In its simplest form, you can create a mock `Claim` entity using Java code such as:

```
import com.guidewire.cc.external.entity.Claim;
...
myClaim = EntityMock.newEntityMock(Claim.class);
```

That type of mock entity supports basic getter and setter methods.

However, suppose you want to test a `MessageTransport` plugin interface implementation, including the acknowledgements. Suppose you want to capture method calls to `message.reportAck(...)` and confirm that the call happened. You can add a `reportAckCalled` method to the `Message`. Your version of the `reportAck` method set some internal flag and your new `reportAckCalled` method returns the status of the flag. You can use that to if any code actually called `reportAck()`.

To do this, first create a mock `Message` entity interface that extends `com.guidewire.cc.external.entity.Message`. Next, call this new interface `MessageMock`. Declare any **new** methods that you want the delegate to intercept, in this case your custom method `Message.reportAckCalled()`. In your new file `MessageMock.java`:

```
package examples.plugins.messaging;

import com.guidewire.cc.external.entity.Message;

// Mock interface for testing that extends the entity com.guidewire.cc.external.entity.Message
public interface MessageMock extends Message {
  boolean reportAckCalled();
```

Next, create a delegate class that implements these two methods `reportAck` and `reportAckCalled`. This class does **not** implement the `MessageMock` interface. This class only implements the method that you added (the `reportAckCalled` method) and an existing method on `Message` (the `reportAck` method)

The following code calls the new class `MessageDelegate`:

```
package examples.plugins.messaging;

/**
 * Extension delegate that implements testing methods for your
 * {@link MessageMock} instance.  Tests create an
 * implementation of the MessageMock interface using
 * {@link com.guidewire.external.entity.EntityMock}, passing in an
 * instance of this delegate.  Any calls to message.reportAck()
 * or message.reportAckCalled() will be handled by the delegate
 * and will override the basic implementation generated by
 * EntityMock.
 */
public class MessageDelegate {
  private boolean _reportAckCalled = false;

  // intercept the built-in domain method
  public void reportAck() {
    _reportAckCalled = true;
  }

  // create a custom method on the mock entity
  public boolean reportAckCalled() {
    return _reportAckCalled;
}
```

Finally, in your test code, create a mock `Message` entity and assign your delegate object as the delegate. The delegate object intercepts any method requests and runs the delegate's methods instead. You can naturally test your message transport plugin code that uses the `Message` entity without running as an actual plugin in a ClaimCenter server:

```
// Create a new message to send.  Note the use of a MOCK implementation of the Message entity
MessageMock message = (MessageMock) EntityMock.newEntityMock(MessageMock.class);

// Add a delegate object that handles entity methods
MessageDelegate delegate = new MessageDelegate();
EntityMock.addDelegate(message, delegate);

// set an ID (the primary key for this entity) -- this might not be required in all cases, but
// it is good practice to set it in case any code tries to check entity.getID()
EntityMock.setId(message, 1);

// Create an instance of the transport plugin. This is test code, but in real Java plugin usage,
// the ClaimCenter server would create this internally in the ClaimCenter JVM!
MessageTransportImpl transport = new MessageTransportImpl();

// Send the message, which must put it on the queue and then acknowledge the message.
transport.send(message, message.getPayload());

// Block until the queue has emptied and listeners have been notified
```

```
// This example assumes that the queue might be multithreaded
queue.waitUntilEmpty();

// USE THE CUSTOM METHOD on the mock Message Object to test it!!!!!
assertTrue(message.reportAckCalled());
```

Modifying EntityFactory for Mock Entities

As mentioned earlier, in its simplest form, you can create a mock `Claim` entity using Java code such as:

```
import com.guidewire.cc.external.entity.Message;
...
myClaim = EntityMock.newEntityMock(Claim.class);
```

Alternatively, you can configure the entity library's default entity factory to create only mock entities. For example, this would allow you to test code that generates a complex graph of entities without creating real entities that persist in the ClaimCenter database.

This would cause all code that changes the behavior of calls such as the following:

```
EntityFactory.getInstance().newEntity("Claim")
```

Code like that now return mock entities as if it had called:

```
myClaim = EntityMock.newEntityMock(Claim.class);
```

To make this change, include this command in your test code:

```
EntityMock.initEntityFactory();
```

For more information about the `EntityFactory`, see "Java Entity Utility APIs" on page 121.

chapter 7

Query Builder

Gosu provides an advanced query system to search your application database for business data entities. Its structure is similar to SQL, which makes your queries easy to write and optimize. This topic presupposes that you have a basic understanding of database concepts and terminology. Database design and database query design and optimizations are broad topics outside the scope of this documentation. Refer to books specifically covering SQL query optimization for more information on these topics.

IMPORTANT For new code, Guidewire **strongly recommends** that you use the newer query builder APIs rather than the older system called *find expressions*. For more information on the older `find` expression APIs, see "Find Expressions" on page 159.

This topic makes extensive use of the following related topics:

- "Gosu Generics" on page 221
- "Gosu Blocks" on page 213

This topic includes:

Query Builder Overview

Gosu provides an advanced query system to search your application database for business data entities. Its structure is similar to SQL, which makes your queries easy to write and optimize.

The most important part of this to remember is that three important classes in the system all implement an interface called `IQueryBuilder`. Many of the important methods you might want to call are defined on that interface, including the common methods `select` and `join`. This means that to assemble a complex query from different elements, you can often chain them together on a single line. You need not worry about whether the method result was a query, a table, or a restriction. ClaimCenter implements many of these methods as Gosu enhancements on the `IQueryBuilder` type. Refer to the enhancement `IQueryBuilderEnhancement` for details.

IMPORTANT Many important query methods are defined as enhancements on all types that implement `IQueryBuilder`, which includes all queries, tables, and restrictions. This makes it easier to **chain** together different actions on one line of Gosu code, which can result in more concise and easier to understand code.

The query builder system consists of the parts described in the following table, each row representing a different class in the `gw.api.database` package.

What	Implemented by	Extends from	Implements IQueryBuilder	Description
queries	`Query`	`Table`	Yes	An abstraction of a query. The first step in using these query builder APIs is to make a query object. Next, you call methods on the query, some of which are defined in the superclasses `Restriction` and `Table`. In addition to the `IQueryBuilder` enhancement, the enhancement class `QueryEnhancement` adds the `make` method. The `make` method is crucial for creating a query initially (before combining with predicates, restrictions, or other queries). Note that this class implements `IQueryBuilder` and most common methods are enhancements on `IQueryBuilder`. This means that if you want to chain predicates and queries together, then you typically can do so without thinking of the exact type.
tables	`Table`	`Restriction`	Yes	A logical database table (in contrast to an actual database table). The enhancement class `TableEnhancement` adds some methods in Gosu. If you use one of the join methods off of a query, its result is a Table, not a Query. Similarly, if you use the advanced `HAVING` clause feature, the `having` method returns a table. Note that this class implements `IQueryBuilder` and most common methods are enhancements on `IQueryBuilder`. This means that if you want to chain predicates and queries together, then you typically can do so without thinking of the exact type.
restrictions	`Restriction`	*none*	Yes	This root class contains various query *predicate* methods. Predicate methods compare database columns to fixed values, values in other columns, or certain types of functions that aggregate data from other properties. For example, this class contains the common methods `compare` and `between`. The enhancement class `RestrictionEnhancement` adds some methods in Gosu. `Restriction` has other subclasses for Boolean predicates: `OrRestriction` and `AndRestriction`. Note that this class implements `IQueryBuilder` and most common methods are enhancements on `IQueryBuilder`. This means that if you want to chain predicates and queries together, then you typically can do so without thinking of the exact type.
query results	`IQueryResult`	*an internal class*	No	Represents a query result, which is what you get from the result of the `query.select(...)` method. This object implements the Iterable interface, so you can use it with a `for` loop. The enhancement class `QueryResultEnhancement` adds some methods in Gosu.
database functions	`DBFunction`	*an internal class*	No	A container class for database functions, which are static methods off of the `DBFunction` class. The enhancement class `DBFunctionEnhancement` adds some methods in Gosu.

Basic Queries

A query object (the `Query` class) represents a query on a table to retrieve results. Queries operate on a primary table, which you specify as you create the query. You can link multiple predicate restrictions together with logical AND and OR predicates. You can link multiple queries together or union and intersect commands. Both of those features are discussed later.

To make a simple query, use the query class's static method called `make`. Pass the `make` method the entity type you want to search. Each result of the query has this entity type. Gosu uses the Gosu generics feature to include which type you are searching for, and Gosu uses generics and type inference so your code can be easy to understand.

For example, to search the `User` table, first create a query and pass the entity type to the `make` method:

```
uses gw.api.database.Query

var q = Query.make(User)
```

Note: The variable `q` has the parameterized type `Query<User>`. This is generics notation meaning "a query, abstracted to work with the `User` type". For more information about generics, see "Gosu Generics" on page 221.

Next, add some sort of test or limitation. For example, you might want to match a field to a certain value or a another type of comparison. Or, you could test whether a property has a specific value or value greater than 100. To add these tests, call methods on the query object that specify the type of comparison. The comparison methods and the `make` method previous mentioned are all implemented using Gosu enhancements on `Query` and `Restriction` classes. Refer in Studio to the source to the enhancement classes such as `QueryEnhancement` and `RestrictionEnhancement` in the package `gw.api.database`.

The following example shows adding a test for an equality comparison of the `PublicID` property:

```
q.compare("PublicID", Equals, "systemTables:1")
```

This compares the `PublicID` property with the value `"systemTables:1"` and matches that row if they are equal. The middle argument in the method call specifies the type of comparison. The complete list of predicate methods and argument options are in the table later in this section.

To extract the items, the simplest way is to iterate across results using a `for` loop with the result of `q.select()`. For example:

```
for (a in q.select()) {
  print(a)
}
```

The result of `q.select()` is a query result, which implements the `Iterable` interface. This allows you to use query results with a `for` loop, as shown in the previous example.

IMPORTANT The `select` method is defined on the `IQueryBuilder` interface. Queries implement this interface, but so do tables and restrictions objects. This is important because building up complex queries typically involve methods that return one of these other two types of objects (tables and restrictions). You can optionally write your code such that you can take the result of one action and directly call the `select` method at the end. This type of direct builder pattern is called *chaining*. You can use chaining with any of the methods defined on `IQueryBuilder`.

You can use any of the enhancement properties and methods defined on the `Iterable` interface. For example, if you want to just get the first item, simply call the enhancement property `First` on this `Iterable` object:

```
var u = q.select().FirstResult
```

For much more information about extracting, filter, and ordering results, see "Returning Query Results" on page 143.

The following table lists the types of comparisons and matches you can make with methods on the query object.

Predicate method	Arguments (Type)	Description

`compare`	• column name (`String`)
	• operation type (`Relop`)
	• value (`Object`)
		Compares a column to a value. For the operation type, pass one of the following values to represent the operation type:
		• `Equals` - Matches if the values are equal
		• `NotEquals` - Matches if the values are not equal
		• `LessThan` - Matches if the row's value for that column is less than the value passed to the `compare` method.
		• `LessThanOrEquals` - Matches if the row's value for that column is less than or equal to the value passed to the `compare` method.
		• `GreaterThan` - Matches if the row's value for that column is greater than the value passed to the `compare` method.
		• `GreaterThanOrEquals` - Matches if the row's value for that column is greater than or equal to the value passed to the `compare` method.
		Pass these values *without* quote symbols around them. These names are values in the `Relop` enumeration.
		For the value object, you can use numeric types, `String` types, Guidewire entities, keys, or typekeys. For `String` values it may be more appropriate in some cases to use the `startsWith` or `contains` methods and to do case insensitive searches. (These other methods are mentioned in later rows in this table.)
		Example of a simple *equals* comparison:
		`q.compare("Priority", Equals, 5)`
		Example of a simple *less than or equal to* comparison:
		`q.compare("Priority", LessThanOrEquals, 5)`
		To compare the value to the value in another column, generate a column reference and pass that instead. See "Column References" on page 136 for extended discussion on this subject.
		You can use algebraic functions that evaluate to an expression that can be evaluated at run time to be the appropriate type. For example:
		`var prefix = "abc:"      // string variable...`
		`var recordNumber = "1234"`
		`q.compare("PublicID", Equals, prefix + recordNumber)`
		Or combine a column reference and algebraic functions:
		`q.compare("Priority", Equals, q.getColumnRef("OldPriority") + 10)`

`compare`	• a range (a `Range` subclass)	Compares the row value for that column to a specified range. The `Range` class is a superclass (an abstract type) with several subclasses that implement different type of ranges. *Refer to the following three rows which list the range subclasses.*
`compare`	• a number range (`NumberRange`)	If the column type is a number (its type extends from `Number`), use the range subclass `NumberRange`. This is very similar to the `ValueRange` subclass. However with `NumberRange`, you do not need to add the generics parameterization syntax (do not add the type name in angle brackets). The following code defines a range of numbers: `q.compare("Priority", new NumberRange(5,10))` To leave the upper or lower bound empty (unbounded), pass `null` for one (but not both) date arguments to the constructor. However, this technique requires that the column permit a `null` value (the column is not non-nullable).

Predicate method	Arguments (Type)	Description
`compare`	• date range (`DBDateRange`)	If the column type is a date, use the range subclass `DBDateRange`. It takes two dates and then a Boolean value that specifies whether to normalize the dates. To normalize a date is to set its time component to midnight of that day. The following code defines a date range. `q.compare("DateOfBirth", new DBDateRange(new DateTime(1950,1,1),new DateTime(1980,1,1), true))` To leave the upper or lower bound empty (unbounded), pass `null` for one (but not both) date arguments to the constructor. However, this technique requires that the column permit a `null` value (the column is not non-nullable).

`compare`	• a value range. Use with text values or anything else other than date or number (`ValueRange`)	For a general value range including `String` comparisons, use the `ValueRange` subclass and specify the type of value in generics notation: `ValueRange<`*`Typename`*`>`. For example `ValueRange<String>`. The following code defines a range of `String` values to match:
		`q.compare("PublicID",`
		`  new ValueRange<String>("abc:1","abc:99"))`
		To leave the upper or lower bound empty (unbounded), pass `null` for one (but not both) arguments to the constructor. However, this technique requires that the column permit a `null` value (the column is not non-nullable).

`between`	• column name (`String`) • start value (`Object`) • end value (`Object`)	Checks whether a value is between two values. This is functionally very similar to the `compare` method when used with a range argument. This method supports `String` values, date values, and number values, just as the range version method `compare`. `q.between("PublicID", "abc:01", "abc:99")` Unlike using the `compare` method with the range option, the between method does not support unbounded queries. In other words, you must always supply non-null values to the `between` method.
`compareIn`	• column name (`String`) • list of values that could match the database row for that column (`Object[]`)	Compares the value for this column for each row to a list of objects that you specify. If the column value for a row matches **any** of them, the query successfully matches that row. For example: `q.compareIn("PublicID", {"default_data:1", "default_data:3"})`
`compareNotIn`	• column name (`String`) • list of values that could match the database row for that column (`Object[]`)	Compares the value for this column for each row to a list of objects that you specify. If the column value for a row matches **none** of them, the query successfully matches that row. For example: `q.compareNotIn("PublicID", {"default_data:1", "default_data:3"})`
`startsWith`	• column name (`String`) • substring value (`String`) • ignore case (Boolean)	Checks whether the value in that column for each row starts with a specific substring. For example, if the substring is `"jo"`, it will match the value `"john"` and `"joke"` but not the values `"j"` or `"jar"`. If you pass `true` to the Boolean argument (the third argument), Gosu ignores case differences in its comparison. For example: `q.startsWith("FirstName", "jo", true /* ignore case */)` Note: If you choose case sensitivity, Gosu uses the denormalization functions for internationalization. Gosu generates a SQL LOWER function to implement case sensitivity if you choose it. However, if the data model definition for the column has its `supportsLinguisticSearch` attribute set to `true`, Gosu uses the denormalized column instead of the SQL `LOWER` function.
`contains`	• column name (`String`) • contains value (`String`) • ignore case (Boolean)	Checks whether the value in that column for each row contains a specific substring. For example, if the substring is `"jo"`, it will match the value `"anjoy"` and `"job"` but not the values `"yo"` or `"ji"`. If you pass `true` to the final argument, Gosu ignores case differences in its comparison. For example: `q.contains("FirstName", "jo", true /* ignore case */)`
`subselect`	See "Advanced Queries (Subselects and Joins)" on page 150.	Perform a join with another table and select a subset of the data by combining tables. For more information, see "Advanced Queries (Subselects and Joins)" on page 150.

Examples

The following is a simple but complete example you can paste into the Gosu Tester when connected to a running server. It queries for a `User` entity with a specific public ID. Since entity public IDs must be unique, this query returns at most a single record. We use the `AtMostOneRow` enhancement property to ensure that this assumption is correct:

```
uses gw.api.database.Query

var userID = "systemTables:1" // search for this public ID

var q = gw.api.database.Query.make(User)

q.compare("PublicID", Equals, userID)

var u = q.select().AtMostOneRow

if (u != null) {
  print("The user is " + u)
}
else {
  print("No such user public ID.")
}
```

This is another opportunity to use chaining for more concise code. You can rewrite this code as the following:

```
uses gw.api.database.Query

var userID = "systemTables:1" // search for this public ID

// chain multiple actions together
var u = Query.make(User).compare("PublicID", Equals, userID).select().AtMostOneRow

if (u != null) {
  print("The user is " + u)
}
else {
  print("No such user public ID.")
}
```

IMPORTANT The `select` and `compare` methods are defined on the `IQueryBuilder` interface. Queries implement this interface, but so do tables and restrictions objects. This is important because building up complex queries typically involve methods that return one of these other two types of objects (tables and restrictions). You can optionally write your code such that you can take the result of one action and directly call the `select` method at the end. This type of direct builder pattern is called *chaining*. You can use chaining with any of the methods defined on `IQueryBuilder`.

Using Comparison Predicates with Null Values

In a relational database, you can define columns that allow null values and ones that require every row to have a value. In a Guidewire application database, you can define entity properties that allow null values and ones that require every instance to have a value.

Selecting Instances Based on Null or Non-Null Values

Use the `compare` method with the `Equals` or `NotEquals` operator to select entity instances based on null or non-null values. The following Gosu sample code returns all `Person` instances where the birthday is unknown.

```
uses gw.api.database.Query

var query = Query.make(Person)
query.compare("DateOfBirth", Equals, null)
```

The following Gosu sample code returns all `Address` instances where the first address line *is* known.

```
uses gw.api.database.Query

var query = Query.make(Address)
query.compare("AddressLine1", NotEquals, null)
```

How Null Values Get in the Database

Null values get in the database only for entity properties that the *Data Dictionary* defines as `non-null`. To assign null values to entity instance properties, use the special Gosu value `null`. The following Gosu sample code sets an `int` property and a `datetime` property to `null` on a new entity instance.

```
var aPerson = new Person()

aPerson.DateOfBirth = null      / -- set a datetime to null in the datbase
aPerson.NumDependents = null    / -- set an int to null in the database
```

After the bundle with the new `Person` instance commits, its `DateOfBirth` and `NumDependents` properties are null in the database.

Blank and Empty Strings Become Null in the Database

To assign null values to `String` properties, use the special Gosu value `null` or the empty string (`""`). If you set the property of an entity instance to a blank or empty string, ClaimCenter coerces the value to `null` when it commits the instance to the database.

The following Gosu sample code sets three `String` properties to different values.

```
var anAddress = new Address()

anAddress.AddressLine1 = " "    / -- set a String to null in the database
anAddress.AddressLine2 = ""     / -- set a String to null in the database
anAddress.AddressLine3 = null   / -- set a String to null in the database
```

However, after the bundle with the new `Address` instance commits, all three address lines are `null` in the database. Before ClaimCenter commits `String` values to the database, it trims leading and trailing spaces. If the result is the empty string, it coerces the value to `null`.

Note that for non-null `String` properties, you must provide at least one non-whitespace character. You cannot work around a non-null requirement by setting the property to a blank or empty string.

See also

- “String Variables Can Have Content, Be Empty, or Be Null” on page 47 in the *Gosu Reference Guide*

Controlling Whether ClaimCenter Trims Whitespace Before Committing String Properties

You can control whether ClaimCenter trims whitespace before committing a `String` property to the database with the `trimwhitespace` column parameter in the data model definition of the `String` column. Columns that you define as `type="varchar"` trim leading and trailing spaces by default.

To prevent ClaimCenter from trimming whitespace before committing a `String` property to the database, include the `trimwhitespace` column parameter in its column definition, and set the parameter to `false`.

```
<column
    desc="Primary email address associated with the contact."
    name="EmailAddress1"
    type="varchar">
    <columnParam name="size" value="60"/>
    <columnParam name="trimwhitespace" value="false"/>
</column>
```

The parameter controls only whether ClaimCenter trims leading and trailing spaces. You cannot configure whether ClaimCenter coerces an empty string to null.

Viewing the SQL for a Query

To print an approximation of the SQL for the query, call the query’s `toString` method.

```
uses gw.api.database.Query

var q = Query.make(User).compare("PublicID", Equals, "systemTables:1")

print(q.select().toString())
```

The output might look something like the following:

```
[4, 13, 15, 5, 9, 1, systemTables:1, default_data:1]
SELECT  FROM cc_contact qRoot WHERE qRoot.Subtype IN (?,?,?,?,?,?)
  AND qRoot.PublicID IN (?,?) AND qRoot.Retired = 0
```

The SQL this method returns might not be precisely what Gosu sends to the database. There might be differences due to internal database optimization.

You can also request that the application to log any SQL that it actually sends to the database. To do this, call the `withLogSQL` method on a query. It takes a single `boolean` argument that specifies whether to log the query to the system logs. If you set this to `true`, the SQL prints to the standard out and also query logs to the system logs in logging category `Server.Database`. If you set this to false, the query prints the query but does not log it to the system logs.

Printing and logging does **not** happen immediately when you call the `withLogSQL` method. Instead, SQL printing and logging happen later when your code uses other APIs that execute the query and send SQL to the database.

Given a `Query` object `q`, you enable logging with code such as:

```
q.withLogSQL(true)
```

IMPORTANT The printing and logging occur when you run the query using other query APIs, not at the time you call the `withLogSQL` method.

Column References

Earlier examples compared a property in an entity to a **value** that you provide directly in the method call. Alternatively, you can compare a property to the value in another column.

For predicates whose method signatures require a value instead of a column, you can get a reference to the desired database column and use that result in the predicate. To get a *column reference*, use the `getColumnRef` method on a query object (or a table object or a restriction object). Pass that method the column name as a `String` value. Next, pass the column reference result to one of the comparison or match predicates *instead* of a value to compare against. The `getColumnRef` method is defined on queries, tables, and restrictions

For example, suppose you want to compare the value in one property with the value in another property using the compare method. Do not pass a value, such as 12 or `"John Smith"`, to the predicate method. Instead, get the column reference for the column using `getColumnRef` and provide the result of that method as the value in the third parameter.

For example, suppose we want to search for all contacts with different values in each row's `EmailAddress1` and `EmailAddress2` properties. Your code might look like the following:

```
uses gw.api.database.Query
var q = Query.make(Contact)
q.compare("EmailAddress1", NotEquals, q.getColumnRef("EmailAddress2"))

for (a in q.select()) {
  print("public ID '" + a.PublicID + " with name " +
    a.DisplayName + " and emails " + a.EmailAddress1 + "," + a.EmailAddress2)
}
```

Notice that you do not need to construct and pass a column reference to the **first** argument to the `compare` method. The method signatures for the comparison predicates take the column name directly without need for conversion with `getColumnRef`.

Note: The result of `getColumnRef` does not implement the `IQueryBuilder` interface. This means you cannot chain together the result of this method with other objects, like you can for some methods.

Database Functions

Real-world database queries sometimes require native database functions found in SQL. For example, a query might need the power of SQL's native minimum and maximum functions that examine rows already in the database.

The query builder includes built-in database functions implemented as methods on the `DBFunction` class. Each time you use these database functions, Gosu generates exactly one corresponding native database function in the SQL. For example, if you use the `Min` database function to calculate the minimum value in that column, Gosu generates the `MIN` SQL function.

The Gosu database functions all take a single argument, which is a column reference. Use the query method `getColumnRef` and pass it the column name as a `String` value.

You can only use these database functions in the contexts permitted by the SQL standard. For example, all functions listed in the table except for `Constant` represent collective data from multiple rows. In database terms, these functions *aggregate* data from other rows. This functions are *aggregating database functions*. You cannot use an aggregating database function on one column and then compare the function result with the **same** database column or even other columns on the same table. Gosu throws exceptions at run time if you violate SQL-enforced usage limitations of aggregated functions.

IMPORTANT Due to restrictions in the SQL standard, you cannot compare one column to aggregated data in the same table. However, if you use a join or subselect, you can compare one column to another table's aggregated data for its columns.

For example, the following query is valid:

```
q.having().compare(DBFunction.Constant(44), GreaterThanOrEquals,
  DBFunction.Avg(q.getColumnRef("IntegerExt")))
```

The following query is **invalid** because the function compares the column name A an aggregate version of a column from the same table:

```
q.compare("A", Equals, DBFunction.Min(q.getColumnRef("A")))
```

This is also **invalid** even if you use different columns on the same table:

```
q.compare("A", Equals, DBFunction.Min(q.getColumnRef("B")))
```

You can work around this SQL limitation using one of the following approaches:

- Performing a *join* on the table. For information about joins and examples of using database functions with joins, see “Advanced Queries (Subselects and Joins)” on page 150.
- Use the database function in a `having` clause, which you create with the `having` method (see the following example)

For example, the following code performs a join with another table and compares the value in one column in one table with aggregate data in another table:

```
var q = Query.make(SampleBean)
var childTable = q.join(SampleChildBean, "parent")
q.having().between("TextField1", DBFunction.Min(childTable.getColumnRef("TextField1")),
    DBFunction.Max(childTable.getColumnRef("TextField1")))
```

The following table lists the supported database functions. In the example column, note that the symbol `q` refers to an instance of a `Query` object or a `Table` object.

Function	Description	Example
`Min`	Minimum of the values in that column.	`q.compare("A", GreaterThan, DBFunction.Min(q.getColumnRef("B")))`
`Max`	Maximum of the values in that column.	`q.compare(DBFunction.Constant(44), LessThan, DBFunction.Max(q.getColumnRef("B")))`
`Avg`	Average of the values in that column.	`q.compare(DBFunction.Constant(44), GreaterThan, DBFunction.Avg(q.getColumnRef("B")))`
`Count`	Total count of the rows in that column matching the table.	`q.compare(DBFunction.Constant(44), GreaterThan, DBFunction.Count(q.getColumnRef("B")))`
`Sum`	Sum of the rows in that column matching the table.	`q.compare(DBFunction.Constant(44), GreaterThan, DBFunction.Sum(q.getColumnRef("B")))`

Function	Description	Example

`Constant`	For use in a `HAVING` clause, returns a constant wrapped so that it can be used in SQL. For example, in a compare predicate method that normally takes a column name as a `String`, you can instead provide a constant using this function.	An advanced example with a join and `HAVING` clause. This permits a *constant* in the first argument of `compare` instead of a column name:
		`var q = new Query(TestE)`
		`var aTable = q.join(Child, "E")`
		`q.having().compare(DBFunction.Constant("44"), Equals, DBFunction.Count( aTable.getColumnRef( "A" ) ))`
		For more about joins, see "Advanced Queries (Subselects and Joins)" on page 150.

`Expr`	Returns a function defined by a list of column reference and character sequences. See "Example of 'Expr' Database Function" on page 138.	See "Example of 'Expr' Database Function" on page 138.

Example of 'Expr' Database Function

The `Expr` database function returns a function defined by a list of column reference and character sequences. The argument to this database function is a list containing only objects of type:

- `java.lang.CharSequence` - for example, pass a `String` containing SQL operators or other functions
- a column reference - the return result of `query.getColumnRef(`*`columnName`*`)`

Gosu concatenates the objects in the list, in order, to form a SQL expression.

For example, the following example creates a new database function from column references to two columns, with the sum (" + ") operator. You can then use the new function to compare values against the sum of these two columns.

```
// SETUP our database to test our Expr() function example. DO NOT USE IN PRODUCTION SERVERS
// for this example, we populate the SCORE and IntegerEext property in Contacts
var b = gw.transaction.Transaction.getCurrent()
var setupQuery = Query.make(Contact)
for (a in setupQuery.select() index i) {
  var writableEntity = b.add(a)
  writableEntity.Score = i      // set the score to be 1, then 2, then 3, and so on
  writableEntity.IntegerExt = 3 // for this demo, set to a constant
}
b.commit()  // permanently change the database with this data. for non-production use only.



// create a new SQL function that sums two integer properties
var q1 = Query.make(Contact)
var e1 = DBFunction.Expr({q1.getColumnRef("Score") , " + ", q1.getColumnRef("IntegerExt") } )
print("Rows where 6 > Score + IntegerExt")
q1.compare(DBFunction.Constant(6), GreaterThan, e1)
for (a in q1.select()) {
  print("public ID  '" + a.PublicID + " Score " + a.Score + ", IntegerExt " + a.IntegerExt)
}


// create a new SQL function that sums two integer properties
var q2 = Query.make(Contact)
var e2 = DBFunction.Expr({q2.getColumnRef("Score") , " + ", q2.getColumnRef("IntegerExt") } )
print("Rows where 6 <= Score + IntegerExt")
q2.compare(DBFunction.Constant(6), LessThanOrEquals, e2)
for (a in q2.select()) {
  print("public ID  '" + a.PublicID + " Score " + a.Score + ", IntegerExt " + a.IntegerExt)
}
```

This prints results similar to the following:

```
Rows where 6 > Score + IntegerExt
public ID  'systemTables:1 Score 0, IntegerExt 3
public ID  'default_data:1 Score 1, IntegerExt 3
public ID  'default_data:2 Score 2, IntegerExt 3
Rows where 6 <= Score + IntegerExt
public ID  'default_data:3 Score 3, IntegerExt 3
public ID  'test:5 Score 4, IntegerExt 3
```

```
public ID  'test:6 Score 5, IntegerExt 3
public ID  'test:7 Score 6, IntegerExt 3
public ID  'test:8 Score 7, IntegerExt 3
public ID  'test:9 Score 8, IntegerExt 3
public ID  'test:10 Score 9, IntegerExt 3
public ID  'test:11 Score 10, IntegerExt 3
public ID  'test:12 Score 11, IntegerExt 3
public ID  'test:13 Score 12, IntegerExt 3
public ID  'test:14 Score 13, IntegerExt 3
public ID  'test:15 Score 14, IntegerExt 3
public ID  'test:16 Score 15, IntegerExt 3
public ID  'test:17 Score 16, IntegerExt 3
```

Combining Queries and Predicates

You can combine multiple predicates in a single query in several ways:

- Simple 'AND'
- Union and Intersection of Queries
- Boolean Algebra of Predicates

Simple 'AND'

The simplest way to combine multiple predicates is to combine the predicates with a logical AND. This means the query finds all rows that satisfy **all predicates**. To link predicates with simple logical AND, simply call the predicate methods (such as `compare` or `between`) repeatedly on the query on separate lines. Each call to the predicate method adds the predicate in series, by default implicitly linked by a logical AND.

For example, you want to return rows that satisfy **all** of the following predicates:

- the `Priority` property has value greater than 11
- the `ItemNum` property has value less than 33
- the `DivisionNumber` property has value greater than 6

Call the predicate methods as separate commands to add them to the query. After you add a predicate, by default Gosu adds a logical AND between the requirements already added and the new predicate.

```
var q1 = Query.make(MyEntity)
q1.compare("Priority", GreaterThan, 11)
q1.compare("ItemNum", LessThan, 33)
q1.compare("DivisionNumber", GreaterThan, 6)
```

The result query returns only records that satisfy all three predicates.

Alternatively, you can link them in series, calling a predicate method **on the result of the previous predicate**. The following example does this and is functionally equivalent to the previous example:

```
var q1 = Query.make(MyEntity)
q1.compare("Priority", GreaterThan, 11).compare("ItemNum", LessThan, 33).compare(
  "DivisionNumber", GreaterThan, 6)
```

IMPORTANT The default linking behavior of predicates on a query or table is a logical AND. This is true whether you run the predicate methods on separate lines of Gosu code or link them in series. If you need a logical OR, refer later in this topic to "Boolean Algebra of Predicates" on page 140.

Union and Intersection of Queries

You can create a query that is a union or intersection of two separate queries by using the `union` and `intersect` methods on queries, respectively. These methods generate and return a new query object. Save that new query object and iterate across that query instead of the original queries. Compare and contrast the following:

- A query union contains all rows that are found by **either** original query.
- A query intersection contains all rows that are found by **both** original queries.

In both cases, the resulting query returns **no** duplicate rows.

You can add use multiple `union` or `intersect` commands in series to make more complex queries.

Paste the following example code into the Gosu tester to run a union query and intersection query on the `User` table. This example requires the server to include the default users in the sample data. The sample gets these users by public ID. This example creates a union query, an intersection query, and also demonstrates combining union and intersection in one query.

```
uses gw.api.database.Query
uses gw.api.database.IQueryBeanResult

var q1 = Query.make(Person).compareIn("PublicID", {"systemTables:1", "default_data:1"})
var q2 = Query.make(Person).compareIn("PublicID", {"default_data:1", "default_data:2"})

var unionQuery = q1.union(q2)
var intersectionQuery = q1.intersect(q2)
var complexQuery = unionQuery.intersect(q1)


// Print the results (using a block to print multiple queries in a standard way)
var printResults = \ qs : IQueryBeanResult, msg : String -> {
  print(msg)
  for (a in qs)
    print(" public ID " + (a as KeyableBean).PublicID + ", " + a["DisplayName"])
}
printResults(q1.select(), "Query #1")
printResults(q2.select(), "Query #2")
printResults(unionQuery.select(), "Union Query")
printResults(intersectionQuery.select(), "Intersection Query")
printResults(complexQuery.select(), "Complex Query")
```

This code prints the following:

```
Query #1
 public ID default_data:1, Super User
 public ID systemTables:1, Default Owner
Query #2
 public ID default_data:1, Super User
 public ID default_data:2, System User
Union Query
 public ID default_data:2, System User
 public ID default_data:1, Super User
 public ID systemTables:1, Default Owner
Intersection Query
 public ID default_data:1, Super User
Complex Query
 public ID default_data:1, Super User
 public ID systemTables:1, Default Owner
```

Chaining Limitations

There are some important considerations with *chaining* results of these operations. You can chain together predicates or groupings that return a query, table, or restriction, and call `union` or `intersect` on those. From the previous example:

```
var q1 = Query.make(Person).compareIn("PublicID", {"systemTables:1", "default_data:1"})
var q2 = Query.make(Person).compareIn("PublicID", {"default_data:1", "default_data:2"})

var unionQuery = q1.union(q2)
```

However, the result type of a union or intersect operation has a different type than regular queries. Instead of `Query<T>`, the type is `GroupingQuery<i>`.

This class has the same methods as a `Query`. However, it does not extend from `Table` or implement `IQueryBuilder`. This means you cannot modify the predicates or tables involved in a `GroupingQuery` after creating it. For example, you cannot call the `compare` or `between` method on the result of an intersect or union.

Boolean Algebra of Predicates

You can also use advanced Boolean algebra to combine predicates using logical AND and OR. To do this, use the `and` and `or` methods on queries, tables, and restrictions. The `and` and `or` methods modify the original query and add a clause to the query. This feature requires using and understanding the syntax of Gosu blocks. Blocks are special functions that you defined in-line within another function. For more information about blocks, see "Gosu Blocks", on page 213.

As mentioned earlier in this topic, the default behavior of a new query (what you create with the query `make` method) is an implicit *logical AND* between its predicates. This documentation refers to this quality as a *predicate linking mode* of `AND`. A new query always has an implicit *predicate linking mode* of `AND`.

The most important thing to know when using `and` and `or` methods is that these methods can change the default linking mode of a new series of predicates. These methods change only the linking mode of **predicates defined within the block** that you pass to the method. The `and` and `or` methods never change the linking mode of predicates that you already added. Similarly, `and` and `or` methods never change the linking mode of previous predicates compared to the new group of predicates that you are about to add.

You can think of the `or` method conceptually with the following English meaning:

- To the previous specified restrictions, add one parenthetical phrase to the query string.
- Link this new parenthetical phrase to previous restrictions in the linking mode I am already using.
- The block I specify includes a series of predicates.
- **For new predicates inside the block, use a predicate linking mode of OR between them.**
- One of the predicates defined in the block could instead be another "or" or "and" grouping, each created with another usage of `and` and `or` methods.

You can think of the `and` method conceptually with the following English meaning:

- To the previous specified restrictions, add one parenthetical phrase to the query string.
- Link this new parenthetical phrase to previous restrictions in the linking mode I am already using.
- The block I specify includes a series of predicates.
- **For new predicates inside the block, use a predicate linking mode of AND between them.**
- One of the predicates defined in the block could instead be another "or" or "and" grouping, each created using another embedded `and` and `or` methods.

The syntax of the `or` method is

```
query.or( \ OR_GROUPING_VAR -> {
  OR_GROUPING_VAR.PredicateOrBooleanGrouping(...)
  OR_GROUPING_VAR.PredicateOrBooleanGrouping(...)
  OR_GROUPING_VAR.PredicateOrBooleanGrouping(...)
  [...]
}
```

The syntax of the `and` method is

```
query.and( \ AND_GROUPING_VAR -> {
  AND_GROUPING_VAR.PredicateOrBooleanGrouping(...)
  AND_GROUPING_VAR.PredicateOrBooleanGrouping(...)
  AND_GROUPING_VAR.PredicateOrBooleanGrouping(...)
  [...]
}
```

Each use of *`PredicateOrBooleanGrouping`* could be either:

- a predicate method such as `compare` or `between`
- another Boolean grouping by calling the `or` or `and` method again.

Notice that within the block, you call the predicate and Boolean grouping methods on the argument passed to the block itself. Do not call the predicate and Boolean grouping methods on the original query.

In the above examples, *`OR_GROUPING_VAR`* and *`AND_GROUPING_VAR`* refer to a grouping variable name that helps identify the "or" or "and" link mode in each peer group of predicates. You can call these block parameter variables whatever you want. However, Guidewire recommends naming the block parameter variable with names that indicate one of the following:

- **Name reminds you of the linking mode.** Use variables that remind you of the linking mode between predicates in that group, for example `or1` and `and1`. Note that you cannot call the variable simply `or` or `and`, since those are language keywords. Instead, add numbers or other unique identifiers to the word "and" or "or".

- **Name with specific semantic meaning.** For example, use `carColors` for a section that checks for car colors

For example, the following simple example links three predicates together with logical OR. In other words, the query returns rows if **any of the three** predicates are true for that row.

```
var q1 = Query.make(MyEntity)

q1.or( \ or1 -> {
  or1.compare("Priority", GreaterThan, 11)
  or1.compare("ItemNum", LessThan, 33)
  or1.compare("DivisionNumber", GreaterThan, 6)
})
```

For example, the following simple example links three predicates together with logical AND. In other words, the query returns rows if **all three** predicates are true for that row.

```
var q1 = Query.make(MyEntity)

q1.and( \ and1 -> {
  and1.compare("Priority", GreaterThan, 11)
  and1.compare("ItemNum", LessThan, 33)
  and1.compare("DivisionNumber", GreaterThan, 6)
})
```

Note: This is functionally equivalent to simple linking of predicates (the default linking mode is AND):

```
var q1 = Query.make(MyEntity)

q1.compare("Priority", GreaterThan, 11)
q1.compare("ItemNum", LessThan, 33)
q1.compare("DivisionNumber", GreaterThan, 6)
```

The power of the query building system is the ability to combine `AND` and `OR` groupings. For example, suppose we wanted to represent the following pseudo-code query:

```
(Priority > 11) OR ( (ItemNum < 33) AND (DivisionNumber > 6) )
```

The following query represents this pseudo-code query using Gosu query builders:

```
q1.or( \ or1 -> {
  or1.compare("Priority", GreaterThan, 11)
  or1.and(\ and1 -> {
    and1.compare("ItemNum", LessThan, 33)
    and1.compare("DivisionNumber", GreaterThan, 6)
  })
})
```

Notice that the outer "or" contains two items, and the second item is an "and" grouping. This directly parallels the structure of the parentheses in the pseudo-code.

You can combine and populate these "and" and "or" groupings with any of the following:

- predicate methods
- database functions
- subselect operations. For an advanced example using subselect and Boolean operators, see "Advanced Queries (Subselects and Joins)" on page 150.

If your query consists of a large number of predicates linked by `OR` clauses, there is an alternative approach that might perform better in real-world performance, depending on your data:

- If the OR clauses all test a single property, rewrite and collapse the tests into a single compare predicate using the `compareIn` method, which takes a list of values. For more information, see "Basic Queries" on page 131. For example, suppose your query checks a property for a color value to see if it matches one of 30 color values, use `compareIn` for this type of query.
- Otherwise, you might consider creating multiple subqueries and using a `union` clause to combine subqueries. Consider trying the query in both approaches, and then testing real-world performance with large number of records. See "Combining Queries and Predicates" on page 139 for more details about creating union queries.

Chaining Inside AND and OR Groupings

The earlier examples used a pattern that looks like the following:

```
var q1 = Query.make(MyEntity)

q1.or( \ or1 -> {
  or1.compare("Priority", GreaterThan, 11)
  or1.compare("ItemNum", LessThan, 33)
  or1.compare("DivisionNumber", GreaterThan, 6)
})
```

In theory, you could use the chaining features of Gosu query APIs to make it more compact using chaining on one line, rather than using three different lines. However, in this case it might make the code harder to understand.

The naming of the Gosu block variable with `"or"` in the name more closely matches the English construction that would describe the final output. In other words, there is an `"or"` clause between each one. Consider simply using this pattern in the example and avoid chaining the predicates together.

Returning Query Results

To extract rows from a query, call the `select` method on the query (or table, or the result of a chained series of clauses). The `select` method returns a query result object. This is the type `IQueryResult<QT,RT>` in the `gw.api.database` package. The `<QT,RT>` part of that type name is generics notation (see “Gosu Generics” on page 221) that indicates it is parameterized across two dimensions:

- `QT` - the query type. This means the entity type you are searching for, such as `Contact`
- `RT` - the return type for that column

This means that the `IQueryResult` works with many different combinations of query types and return types in a flexible way and yet is statically typed at compile time.

Use the query result in the following ways:

- iterate across results
- order results based on existing database columns (see “Ordering results” on page 147)
- filter the original query
- load only a subset of columns (see “Selecting Columns and Returning Results in Custom Formats” on page 145)
- encapsulate or return each row in any arbitrary format (see “Selecting Columns and Returning Results in Custom Formats” on page 145)
- get the number of rows in the result set as of now. Remember that because databases are dynamic, the number of results might change between now and additional actions on the query. Carefully design your code to support this number changing possibility.

Getting Single Rows

The query result object defines several methods for getting a single item.

To get the first item only, simply call the enhancement property `FirstResult`:

```
var u1 = q1.select().FirstResult
```

If you want to get the first item and ensure there is only one match, call the enhancement property `AtMostOneRow` on this `Iterable` object. If there is more than one match, Gosu throws an exception. However, if there are no matches, this returns `null`. The following example gets one row in this way:

```
var u2 = q2.select().AtMostOneRow
```

To ensure there is exactly one match, instead use the `single` method (not a property). It throws an exception if there no matches, in contrast to the `AtMostOneRow` property:

```
var u3 = q3.select().single()
```

Getting Results With an Iterator

The `IQueryResult<QT,RT>` class implements the `Iterable` interface. This means it provides an iterator (the class `Iterator`). The easiest way to use an iterable object is with a `for` loop. For example:

```
uses gw.api.database.Query

var q = Query.make(Contact)
var count = 0

for (e in q.select()) {
  count++
  print("Result #" + count + " is " + e)
}
```

You can also get the iterator directly from the query result by calling the `iterator` method. For example:

```
uses gw.api.database.Query

var q = Query.make(Contact)
var i = q.select().iterator()

if (i.hasNext()) {
  var n = i.next().DisplayName;
  print("Display Name is " + n)
}
else {
  Throw("We expected at least one Contact")
}
```

Converting to Collections, Lists, Sets, and Arrays

You can convert query result sets to lists, collections, and sets. Converting a query to these types runs the database query and iterates across the entire set. The application pulls all entities into local memory. Because of limitations of memory, database performance, and CPU performance, never do this for queries of unknown size. Only do this if you are absolutely certain that the result set size and the size of the object graphs are within acceptable limits. Be sure to test your assumptions under real-world conditions.

WARNING Converting a query to lists or arrays pulls all the entities into local memory. Only do this if you are absolutely certain the result set size and the size of the objects are small. Otherwise you risk memory and performance problems.

If you need the results as an array, you can convert to a list and then convert that to an array using Gosu enhancement methods.

The following example converts queries to different collection-related types including arrays:

```
var q = Query.make(User)

// in production code, converting to lists or arrays can be dangerous
// due to memory and performance issues. Never use this approach unless you are certain
// the result size is small. In this demonstration, we check the count first.
// In real world code, you would not use this approach with the User table, which
// can be larger than 100 rows.
if (q.select().Count < 100) {

  var rc = q.select().toCollection()
  var rs = q.select().toSet()
  var rl = q.select().toList()
  var ra = q.select().toTypedArray()

  print("Collection: statictypeof " + statictypeof rc + "/ typeof " + typeof rc + "/ size " + rc.Count)
  print("Set: statictypeof " + statictypeof rs + "/ typeof " + typeof rs + "/ size " + rs.Count)
  print("List: statictypeof " + statictypeof rl + "/ typeof " + typeof rl + "/ size " + rl.Count)
  print("Array: statictypeof " + statictypeof ra + "/ typeof " + typeof ra + "/ size " + ra.Count)
```

```
}
else {
 Throw("too many query results to convert to in-memory collections")
}
```

This example prints something like the following:

```
Collection: statictypeof java.util.Collection<entity.User>/ typeof java.util.ArrayList/ size 34
Set: statictypeof java.util.Set<entity.User>/ typeof java.util.HashSet/ size 34
List: statictypeof java.util.List<entity.User>/ typeof java.util.ArrayList/ size 34
Array: statictypeof entity.User[]/ typeof entity.User[]/ size 34
```

Notice the types of the Java-based classes (collection, set, and lists) that the example prints. The compile time types (this is what `statictypeof` returns) are parameterized with the query entity type (in this case, `User`). However, the run time type (what `typeof` returns) does not have the parameterization. This is due to generics differences between Gosu and Java. Java implements generics with *erasure*, which means that at run time the parameterization with the exact type is lost. The parameterization does not exist at run time. This behavior is not true for Gosu. If those classes were Gosu classes, the run-time type would preserve the parameterization.

In contrast, note that the `toTypedArray` method generates an array properly typed (an array of `User`) both at compile time and at run time.

Selecting Columns and Returning Results in Custom Formats

In many situations you only need a few columns from the result instead of the whole entity. You can use a variant of the `select` method to do this. The `select` method is only available off of queries.

Instead of calling the `select` method with no arguments, use an alternate method signature that takes a *block*. A block is an in-line function that you define within another function. For more information about blocks, see “Gosu Blocks”, on page 213. You pass a block that takes an object of the query type and returns whatever you want. The return type can be anything that you specify. Any arbitrary type is fine, so you can choose whatever is most convenient for your program. The type does not need to match the native types for the columns in the database.

The full method signature of this alternate `select` method is the following:

```
function select<RT>(columns : block(row : QT) : RT) : IQueryResult<QT, RT>
```

Note that the function returns an `IQueryResult` parameterized by the original query type, as well as the return type of your block. This allows Gosu to infer the block return type from the type of the query due to how Gosu generics work. You do not need to specify the return type of your block explicitly. For related topics, see “Generics and Blocks” on page 226 and “How Generics Help Define Collection APIs” on page 228.

This alternative select method provides the following important benefits:

- **Gosu only loads the columns you need.** The database returns **only** the columns you access in your block. This reduces memory usage and improves overall server performance.
- **You can assemble arbitrary structures or calculations to return in place of full entities.** Since you are not retrieving the entire entity, you choose how to return each result. For example, you can populate a custom Gosu class containing only the fields you need. Or you can simply return a single field that you need, such as a public ID string.

Gosu determines which columns to load by inspecting the block at compile time (at a low level) to determine which database columns are affected by the query. At compile time, Gosu looks inside the block **without running** it. By detecting that you referenced the `Address1` property on an `Address` entity, Gosu knows that it must load the `Address1` property. Any properties not mentioned at all in your block are not loaded from the database for this query.

As you iterate across the result set, after a row loads from the database, Gosu runs your block.

IMPORTANT Gosu actually inspects the parse tree of the `select` block to determine which properties to load at run time. For each result in the result set, Gosu executes your block and replaces the values of database columns with values from the database.

Gosu imposes the following rules on what kinds of tasks you can do inside the Gosu block:

- All references from the block argument (the database row) must be references to properties backed by actual database columns.

WARNING You cannot use virtual properties, enhancement methods, or other entity methods within the block that you pass to select. If you are unsure which properties are backed by actual database columns, refer to the *Data Dictionary*.

- Any references to a database function, which are static methods on the `DBFunction` class, must contain exactly one column reference. Additionally that column reference can have a cast associated with it, and that cast becomes a database cast in the final output).

For example, to extract two database columns into a hash map mapping a `String` to `Integer`, you could use code similar to the following:

```
var rows = Query.make(MyEntity).select(\row-> new HashMap<String, Integer>() {
  "A" -> row.A,
  "E" -> row.E
})
```

This returns an `IQueryResult<MyEntity, HashMap<String,Integer>>`. The `IQueryResult` class extends `Iterable`, more specifically `IQueryResult<QT,RT>` extends `Iterable<RT>`. This means that this method returns an iterable object of type `Iterable<HashMap<String,Integer>>`.

In the previous example, note the references to `row.A` and `row.E`. In the final result, these references become database references.

The following is a more subtle example that fetches three columns, assuming that MyEntity.E references an entity that contains properties named E1 and E2:

```
var rows = Query.make(MyEntity).select(\row-> new HashMap<String, Integer>() {
  "A_plus_ten" -> row.A + 10,
  "E1_minus_E2" -> row.E.E1 - row.E.E2
})
```

The columns in the following example are more subtle. This fetches three columns from the database:

- `row.A`
- `row.E.E1`
- `row.E.E2`

Note: Note that Gosu permits the line with a computation computes `row.A + 10`. Gosu does not optimize this using the SQL feature called *computed columns*. Gosu query builder does not support computed columns. Instead, Gosu simply loads the columns from the database and performs the calculation in the Gosu block at run time. Be aware of this implementation detail if you implement complex calculations in your block. If you have complex calculations in your block, remember to test and profile performance at run time with large loads to determine the performance impact of your calculations.

The following is a complex example that returns a custom anonymous inner class and initializing two properties (A and E):

```
var rows = Query.make(TestA).select(\row-> {
    if (row.A < 10) {
      return new MyObject() {
        :A row.A + 10,
        :E -> row.E.E1 - row.E.E2
      })
    } else {
      return new MyObject() {
        :A row.A + 1000,
```

```
            :E -> row.E.E1 + row.E.E2
        }
    }
```

Database Functions Within Select Blocks

The following is an example that uses a database function:

```
var rows = Query.make(TestA).select(\row-> new HashMap<String, Integer>() {
      "A" -> DBFunction.SUM(row.A as double),
      "A2" -> row.A2
    })
```

It returns the summary of all `TestA.A`, coerced to `double` to show how to do coercion, and then finally grouped by the property `TestA.A2`.

Within your block, you can use the following database functions: `Sum`, `Max`, `Min`, `Avg`, and `Count`.

Ordering results

Gosu provides methods on the result set objects that order the results in the result set. Gosu uses this information to generate different SQL so the results return in the appropriate order. The database does the ordering and returns records in the correct order. Gosu does not need to load the result into memory to order them. Loading all the entities into memory would often require too much memory and the performance would be poor. Instead, Gosu modifies the SQL that returns the results in the desired order.

Gosu supports multiple levels of sorting on different columns. For example, you could sort an address book table by last name. If last names match for two rows, then sort them by first name. If both first and last name match, sort by birth date. Gosu represents this by a series of methods you call. First, call the method to order by the first desired column, either in ascending order (`orderBy`) or descending order (`orderByDescending`). Next, take the result of that method and specify the second-level sort column by calling its `thenBy` method for ascending order, or `thenByDescending` method for descending order. You can call `thenBy` or `thenByDescending` as many times as desired for third-level sorting, and so forth.

All these methods take a block as their one argument. That block must take exactly one argument, which is the type of the entity for which you are searching. This block is a highly unusual type of block. The Gosu query builder introspects within the block you define and extracts information from it. However, at no time does Gosu actually execute this block. In essence, the block plays the role of a container for you to specify the database column to search. Gosu implements this ordering-related argument as a block because it forces column names to be checked at compile time against the data model.

To specify a database column, use the following syntax for the block:

```
\ row -> row.PROPERTY
```

For example, to specify to sort on the `LastName` property, use the syntax:

```
\ row -> row.LastName
```

Alternatively, you can order by the result of a database function, using the `DBFunction` class. For example:

```
\ row -> DBFunction.Max(row.LastName)
```

Gosu result set objects support the ordering methods listed in the following table.

Method	Description
`orderBy`	Clears all previous ordering, then orders results by the specified column in ascending order.
`orderByDescending`	Clears all previous order methods, and orders by the specified column in descending order.
`thenBy`	Orders by the specified column in ascending order. Does not clear previous orderings.
`thenByDescending`	Orders by the specified column in descending order. Does not clear previous orderings.

The argument to the block is the row. The block must contain the following types:

- A Gosu expression that returns a field path to a simple property. In other words, a period-delimited expression starting with the block variable for the row, and ending in a non-foreign-key database property. An example of a simple version of a field path is `row.LastName`. You can have longer field paths if intermediate property accesses are foreign keys. For example `row.MyChildEntity.AnotherChildEntity.LastName`. Every foreign key property and the final property access must be database-backed properties. They cannot include virtual properties, methods, or any other types of calculations.
- A single legal `DBFunction` reference. You cannot nest database functions or combine them in any way within a single block. (You can, however, include multiple *order by* or *then by* clauses, each of which contains one database function.)

So, for example, the following is legal:

```
var q = Query.make(TestA)
q.compare("A", Equals, 11)
var result = q.select()
result.orderBy(\row-> row.A2).thenByDescending(\row->row.E.E)
```

The following is illegal because mathematical operators such as addition are disallowed within blocks that work with *order by* or *then by* clauses:

```
var q = Query.make(TestA)
q.compare("A", Equals, 11)
var result = q.select()
result.orderBy(\row-> row.A2 + 10 ).thenByDescending(\row->row.E.E + 20 )
```

Locale Sensitivity for Ordering

Ordered database query results automatically uses **locale-sensitive** comparison.

In contrast, note that the collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, those methods do not sort `String` values in a locale-sensitive way.

Found Entities Are Read-only Until Added to a Bundle

Entities that you iterate across after a find query are read-only by default. The entity is loaded in a read-only *bundle*, which is a collection of entities loaded from the database into server memory. By default, find query results are returned in their own read-only bundle separate from the active read-write bundle of any running code.

To change a read-only entity's properties, you must move the entity to a new writable bundle. Typical code adds the entity to the *current bundle* of the running code. For example, if your Gosu code is triggered from a rule set or a plugin method, there is a current bundle that you can access. Simply add each entity to the new bundle to permit changes to the entity and mark the entity as changed. After the entity changes and the new bundle commits successfully, all entity changes are copied to the database. Remember that an entity cannot safely exist in more than one writable (read-write) bundle.

To move the entity to a writable bundle, call the `add` method on a bundle and save the result of the `add` method.

> **IMPORTANT** If you passed a read-only entity to a writable bundle, the return result of the `add` method is a cloned instance of the entity you passed to the method. If you want a reference to the entity so you can make further changes, you must keep a reference to the **return result** of the `add` method. Do not modify the original entity reference. Do not keep a reference to the original entity.

For example:

```
// get the current (ambient) bundle of the runing code
var bundle = gw.transaction.Transaction.getCurrent()

var query = Query.make(TestA)
query.compare("A", Equals, 11)
var result = query.select()
result.orderBy(\row-> row.A2).thenByDescending(\row->row.E.E)
```

```
for( c in query ) {
  bundle.add(c)

  // make changes to one or more properties directly on the entity or its subobjects
  c.MyProperty1 = true
}
```

IMPORTANT For more information about bundles and transactions, see "Bundles and Transactions" on page 275, which discusses the issue and related APIs in much more detail.

Result Counts and Dynamic Queries

A query is always dynamic and returns results that may change if you use the object again. Some results may have been added, changed, or removed from the database from one use of the query object to another use of the query, even within the same function.

The `Count` property of a query gets the current count of items. A common mistake is to rely on the count number remaining constant. That number might be useful in some contexts such as simple user interface display such as "displaying items 1-10 out of 236 results". However, it might be different from the number of items returned from a query even if you iterate across it **immediately** afterward. Some results may add, change, or remove from the database between the time you call the `count` method and when you iterate across it.

Bad Example:

```
// Bad example. Do NOT follow this example. Do NOT rely on the result count staying constant!

// create a query
var query = Query.make(TestA)
query.compare("A", Equals, 11)

// THE FOLLOWING LINE IS UNSAFE
var myArray = new Claim[ query.select().count() ];

For ( x in query.select() index y )
{
   // this line throws out of bounds exception if more results appear since the count was calculated
   myArray[y] = x;
}
```

In most cases, code like this risks throwing array-out-of-bounds errors at run time.

Instead, iterate across the set and count upward, appending query result entities to a `List`.

Good Example:

```
uses java.util.ArrayList

var q = Query.make(User)
q.compare("IntegerExt", NotEquals, 11)

// create a new list, and use generics to parameterize it
var myList = new ArrayList<User>()

var maxResult = 100

for ( x in q.select() index i)
{
   if (i > maxResult) {
     break
  }

  // add a search result to the list
  myList.add(x);
}
```

The important thing to remember is that calling `query.select()` does not snapshot the current value of the result set forever. When you access the `query.select().Count` property, ClaimCenter runs the query but the query contents can change quickly. Database changes could happen in another thread on the current server or on another server in the cluster.

Setting the Page Size For a Query

As you use the result object to iterate **across** the query results, ClaimCenter runs the query. However, Gosu does not typically run the query each time you access the next result in the query. Instead, Gosu automatically loads several results from the query in a batch at one time and caches it for quick access. This means that common actions like iterating across the query are higher performance.

You can customize the number of results to load in each batch. To set the page size, call the `setPageSize` method on the query result object, which is the return value of `query.select()`.

For example:

```
var q = Query.make(User)

// load 10 entities at one time and cache them
q.select().setPageSize(10)
```

Other notes:

- If you plan to modify the entities, see related section “Found Entities Are Read-only Until Added to a Bundle” on page 148.
- In real-world code, be careful not to retrieve too many items and keep references to them. Memory errors and performance problems can occur. Design your code to limit the result set that your code returns.

IMPORTANT Always test database performance under realistic real-world conditions before and after changing any performance tuning settings.

Advanced Queries (Subselects and Joins)

You can use Gosu to query from multiple database tables using a SQL feature called *table joins*. A table join combines records from multiple tables, linking them using columns (each representing entity properties) that exist in more than one table. One table links to another table by containing a column for a foreign key, which is a unique ID column from another table. For example, imagine an address book application with a table of people and each row for a person has a unique ID column. Suppose another table represents students enrolled in courses at a university. The table for classes refer to people only by the unique ID for the person. Consequently, that column exists in both tables. If you want to get a list of names of students in a specific course, you can write a *join query* using information from both tables.

Gosu provides two basic ways to join tables and query against the results. The following table compares and contrasts the two methods, `subselect` and `join`. To decide which API method to use, you must first determine what type of join you need.

The first question is which table contains the foreign key that links the tables. For the sake of this documentation, we refer to the entity you are searching (and want results for) as the *primary table*. The joined table is the *secondary table*.

If the foreign key that links the two tables is defined on the secondary table, this is called a *reverse join*. For example, suppose you search for a set of entities that have child entities:

- If the primary table has a foreign key that link to the secondary table, this is a **forward join**.
- If the secondary table has a foreign key that link to the primary table, this is a **reverse join**.

For joins, there is another choice you need to make. You must choose whether you want to get matches on the primary table even if there is no link to the secondary table at all. For example, suppose you search for a set of entities that have child entities. Do you want results in the primary table, even if the parent does not have a child entity?

- If you do not want an entity to match on the primary table if your inner query on your secondary table does not match items, this is an **inner join**.
- If you want an entity to match on the primary table if your inner query on your secondary table does not match items, this is an **outer join**.

Refer to the following table to determine which method to use. These are general recommendations only. Due to subtleties in how database optimizers work, your performance might vary with different approaches. You might try multiple approaches under realistic loads to determine real-world performance:

	Inner joins	Outer joins
Forward joins	Use `join` method. This method never returns duplicates on the primary table. Because it is an inner join, entities on the primary table only match if there is an associated entity on the secondary table.	Use `outerJoin` method. Forward joins never return duplicates in the primary table. Because it is an outer join, entities on the primary table successfully match (is in the results) if the foreign key to the secondary table is `null`.
Reverse joins	Use `subselect` method, generally speaking. This method never returns duplicates on the primary table. Because it is an inner join, entities on the primary table only match if there is an associated entity on the secondary table. **IMPORTANT:** Every situation is different and in some cases using the `join` method might perform better. For example, if the secondary table including predicates are not very selective. In other words, if the secondary clause returns lots of results, use `join` instead. For important queries, Guidewire recommends trying both approaches under performance testing. If you use `join`, use two-parameter method signature including the table name and column name in the joined table.	Use `join` method two-parameter method signature including the table name and column name in the joined table. Because it is an outer join, entities on the primary table successfully match (is in the results) if the foreign key to the secondary table is `null`. An outer reverse join is susceptible to accidental duplicate rows on the primary table. For example, if you multiple child objects link to the parent, do you want duplicate rows in the parent table for each child that matches. For more discussion on reducing duplicates, see “Duplicates with Reverse Joins” on page 155.

Both these methods create a SQL join and add additional predicates to the `HAVING` part of a SQL query. In contrast, calling comparison and match predicates *directly* on a new query adds predicates to a `WHERE` clause of the SQL query. If you add predicates to the `WHERE` part of a SQL query that is the right hand side of a *join*, Gosu adds the predicates to the SQL query. The new predicates are in the `ON` clause of the join.

For maximum performance and query optimization, only use joins if necessary, and always minimize the number of joins you use. If multiple queries operate on the same table, conceptually you can use joins or intersect operators to perform the query. However, generally speaking it is more efficient to collect those predicates altogether in one clause when possible. For example, suppose one query searches table `A` for a specific value for column `Property1` and another query searches table `A` for a specific value for column `Property2`. Although you could join the two queries together, it is best to create a single query with no joins and just add two compare predicates. That type of database query is easiest to optimize. See “Basic Queries” on page 131 for examples of adding multiple predicates to a single query.

> **IMPORTANT** The `join` method (and its variants) and the `subselect` method are defined on the `IQueryBuilder` interface. Queries implement this interface, but so do tables and restrictions objects. This is important because building up complex queries typically involve methods that return one of these other two types of objects (tables and restrictions). You can optionally write your code such that you can take the result of one action and directly call the `join` or `subselect` method at the end. This type of direct builder pattern is called *chaining*. You can use chaining with any of the methods defined on `IQueryBuilder`.

Using Subselect for Reverse Inner Joins

The `subselect` method reduces the number of rows returned by a query based on a subquery to other queries using a subselect operation. This method is defined on the `Restriction` class. Use `subselect` only if the foreign key that links the two tables is defined on the secondary table. This is called a *reverse join*. For example, suppose you search for a set of entities that have child entities. If the foreign is defined on the child entity to link to the parent, this is a *reverse join*.

> **IMPORTANT** Every situation is different and in rare cases using the `join` method might perform better. For example, if the secondary table including predicates are not very selective. In other words, if the secondary clause returns lots of results, use `join` instead. For important queries, Guidewire recommends trying both approaches under performance testing. If you use `join`, use two-parameter method signature including the table name and column name in the joined table. For more information, see "Join Method" on page 154.

A subselect joins a query or table to another query or table in the following ways:

- Join any column on the parent entity with any entity on the child.
- Join using any function on a column on the parent with any function on any column on the child.
- Join with SQL constant functions. For example, using the SQL `count` constant function:
  ```
  where 0 in (select count(ID) from ...))
  ```

To perform a subselect on a query, call the `subselect` method. There are many different method signatures for this method, which in the following sections are grouped with similar method signatures.

If you have already defined the join table in the query, you can use a set of method signature in the following order:

- a column **in the primary table** *or* a database function
- an operation
- a column **in the join table** *or* a database function

If you pass a column for either the first or third argument, simply pass a `String` version of the column name. The method actually takes a `IEntityPropertyInfo` object, but Gosu does the coercion to this type automatically at compile time. You can also pass a database function, such as `DBFunction.Count(Child, "ID")` or a constant using `DBFunction.Constant(2)`. For more information about database functions, see "Basic Queries" on page 131. For the first and third arguments, you can pass any combination of columns or database functions: both as columns, both as database functions, or one of each in either argument.

For the operation, pass either `CompareIn` or `CompareNotIn`, which are enumeration values of the type InOperation. The query generates a SQL `IN` clause if the operation is `CompareIn`. The query generates a SQL `NOT EXISTS` clause if the operation is `CompareNotIn`.

For example, suppose you have an entity called `SampleParent` with an ID field and some of those objects have subobjects of type `SampleParent` that link back in their Parent field. The following subselect joins these two tables:

```
query.subselect( "ID", CompareIn, "Parent")
```

You can then add additional predicates by calling predicate methods on this object. For example, call the `compare` predicate. See “Basic Queries” on page 131 for more information.

Example with database function:

```
subselect( DBFunction.Constant(2), CompareIn, DBFunction.Count(Child, "ID").
```

Another group of method signatures take an additional argument containing a query that you previously defined. These methods take the following arguments in the following order:

- a column **in the primary table** *or* a database function
- an operation
- a joining query
- a joining column **in the query** *or* a database function

Just like the previous group of method signature variants, simply pass a `String` version of the column name. Similarly, for the first and fourth arguments, you can pass any combination of columns or database functions. You can pass both as columns, both as database functions, or one of each in either argument.

Subselect with Subobjects

For example, suppose we have an entity called `SampleParent` and it has a subobject of entity type `SampleChild`. Suppose the parent object has an `ID` property as a key, and the subobject links to it with its own `Parent` property. To find all `SampleParent` objects that have at least one `SampleChild` subobject in which the child’s `TextField1` has the value `"Hello"`, use the following query:

```
var q = Query.make(SampleParent)
q.subselect( "ID", CompareIn, SampleChild, "Parent").compare("TextField1", Equals, "Hello")
```

This produces the SQL

```
Values=[Hello]
SELECT <beans> FROM cc_sampleparent qRoot WHERE qRoot.ID IN (SELECT qRoots0.parentId
   col0 FROM cc_samplechild qRoots0 WHERE qRoots0.TextField1 = ?)
```

Subselect with Negation (Not In)

In this variant of the previous example, suppose you want to find all `SampleParent` objects that do **not** have `SampleChild` subobjects in which subobject’s `TextField1` property has the value `"Hello"`. You can use the following query:

```
var q = Query.make(SampleParent)
q.subselect( "ID", CompareNotIn, SampleChild, "parent").compare("TextField1", Equals, "Hello")
```

This produces the SQL

```
Values=[Hello]
SELECT <beans> FROM cc_sampleparent qRoot WHERE  NOT EXISTS (SELECT qRoots0.ID col0
  FROM cc_samplechild qRoots0 WHERE qRoots0.parentId = qRoot.ID AND qRoots0.TextField1 = ?)
```

Subselect with Boolean Algebra

The following example combines joins and Boolean algebra:

```
var q = Query.make(TestE)
q.and(\and1->{
  and1.or(\or1->{
    or1.compare("E", Equals, 0)
    or1.compare("E", Equals, 1)
  })
  and1.or(\or2->{
    or2.subselect( DBFunction.Constant(2), CompareIn, DBFunction.Count(Child, "ID")).compare("E",
       Equals, q.getColumnRef("ID"))
    or2.subselect( DBFunction.Constant(3), CompareIn, DBFunction.Count(Child, "ID")).compare("E",
       Equals, q.getColumnRef("ID"))
```

```
        })
      })
```

This produces the SQL:

```
Values=[0, 1]
SELECT  FROM cc_parent qRoot WHERE ((((qRoot.E = ?) OR (qRoot.E = ?))) AND (((2 IN (SELECT
COUNT(qRoots0.ID) col0 FROM cc_test_a qRoots0 WHERE qRoots0.EID = qRoot.ID))
OR (3 IN (SELECT COUNT(qRoots0.ID) col0 FROM cc_test_a qRoots0 WHERE qRoots0.EID = qRoot.ID)))))
```

Join Method

The other Gosu query builder approach for joins is the `join` method defined on all `IQueryBuilder` types: tables, restrictions, and restrictions. The `join` method can do SQL join types that the `subselect` method cannot do. For example, outer joins and self-joins. Additionally, for reverse joins, the `join` method can return duplicate rows on the primary table due to the way SQL natively handles these types of joins.

See “Using Subselect for Reverse Inner Joins” on page 152 for more information about `subselect`. See “Advanced Queries (Subselects and Joins)” on page 150 for more information about the differences. If you have questions about when or how to use the `join` method, contact Guidewire Customer Support.

> **IMPORTANT** The `join` method and its variants are defined on the `IQueryBuilder` interface. Queries implement this interface, but so do tables and restrictions objects. This is important because building up complex queries typically involve methods that return one of these other two types of objects (tables and restrictions). You can optionally write your code such that you can take the result of one action and directly call the `join` method at the end. This type of direct builder pattern is called *chaining*. You can use chaining with any of the methods defined on `IQueryBuilder`.

Forward Joins

If the foreign key is on the primary table linking to the secondary table, use a forward join.

> **Note:** This topic uses the terminology *inner join*, *outer join*, *forward join*, and *reverse join*. For definitions of these terms, see “Advanced Queries (Subselects and Joins)” on page 150.

To perform an inner forward join, first determine the property on the primary table that joins to the secondary table. To join one table to another, call the `join` method and pass the foreign key column as a property name `String` to the `join` method. For example, suppose you have an entity `SampleParent` and each one has a property `MyChildObjectProperty` that contains a foreign key to the `SampleChild` table. The following is an example of a simple inner join:

```
var q = Query.make(SampleParent)
q.join("MyChildObjectProperty")
```

> **Note:** Pass the Gosu version of the property name. Do not pass the actual database column name with the `cc_` prefix, such as `cc_mychildobjectproperty`.

If you do not add additional predicates to the query `q`, the query returns all `SampleParent` entity that contain an associated `SampleChild` entity. Gosu returns one for each match in the `SampleChild` table. (See notes about duplicates later in this topic.)

You can add predicates to the primary table before or after the join. Adding predicates limits the rows that Gosu returns on the primary table:

- If you add predicates to the outer query, the namespace of the properties is by default the properties on the outer table. (For related information, see “Advanced Inline Views” on page 157). For example:
  ```
  var q = Query.make(SampleParent)
  q.compareIn("ID", myEntityIDs)
  q.join("MyChildObjectProperty")
  ```
- If you add predicates to the joined table, also referred to as the inner query, the namespace of properties by default is the set properties on the outer table. However, you can also use columns on the outer table by using

the `getColumnRef` method on the original query. Pass the column name on the primary table to the `getColumnRef`.

If you want to return result on the primary entity table even if there is no associated entity on the secondary table, use the `outerJoin` method instead of `join`.

For example:

```
var q = Query.make(SampleParent)
q.compareIn("ID", myEntityIDs)

// returns a SampleParent entity even if it has no SampleChild
q.outerJoin("MyChildObjectProperty")
```

Reverse Joins

In contrast, in some cases the property that links the object on the secondary table to the primary table exists on the secondary table. This is instead called a *reverse join*. This is the standard design pattern for Guidewire entities in the data model that appear in Gosu as entities properties containing arrays of entities. At the database level, the child objects link to the parent object with a foreign key on the child. Gosu hides this implementation detail by allowing you to view the reverse join as a read-only array on the parent object.

> **Note:** This topic uses the terminology *inner join*, *outer join*, *forward join*, and *reverse join*. For definitions of these terms, see “Advanced Queries (Subselects and Joins)” on page 150.

For example, from Gosu, a claim contains an array of exposures. The exposures link back to the claim with a foreign key on the exposure that links back to the claim. The `Claim.Exposures` property in Gosu hides this implementation detail by allowing you to view the reverse join as a read-only array.

If you want to add a reverse join, you must specify the table you want to join to and which property on the joined table links to the primary table. For example, suppose the child object `SampleChild` contained a property called `Parent` that linked to the parent table.

Call the join method but with the arguments in this order:

- the `Table` to join to (`SampleChild`)
- the column name *on the joined table* (`Parent`)

For example:

```
var q = Query.make(SampleParent)
var childTable = q.join(SampleChild, "parent")
```

If you want to return result on the primary entity table even if there is no associated entity on the secondary table, use the `outerJoin` method instead of `join`.

For example:

```
var q = Query.make(SampleParent)

// returns a SampleParent entity even if it has no SampleChild
var childTable = q.outerJoin(SampleChild, "parent")
```

Duplicates with Reverse Joins

It is critical to note that it is easy to accidentally create duplicate rows on the primary table by accident using reverse joins with the `join` or `outerJoin`. Using the previous example, if more than one `SampleChild` links to the same `SampleParent`, that `SampleParent` is returned twice in the query. One result in the result of the join query exists for each child object of type `SampleChild`.

This duplicate row in the primary table is desirable in some cases. For example, if you intend to iterate across the result and extract properties from each child object that matches the query, this might be what you want.

However, in many cases you likely want to return only maximum one row on the primary table and you must carefully design your query accordingly.

> **WARNING** It is extremely important to understand that reverse joins using `join` and `outerJoin` may generate duplicates on the primary table if there are multiple matches on the secondary table. Design your queries carefully with this in mind. If you do not want duplicates and you want an outer reverse join, consider rewriting to use the `subselect` method approach instead. See "Using Subselect for Reverse Inner Joins" on page 152 for details and examples.

Notice that this duplicate issue for reverse joins does not apply to forward joins. The foreign key to the secondary table is unique in the secondary table. Any entity on the primary table that links to the secondary table on the secondary table in the forward direction contains either:

- `null` (matches no items on the secondary table)
- links to a maximum of one entity on the secondary table.

For reverse joins, try to reduce duplicates on the primary table. The best way is to ensure that the table you join to only has one row that matches the entity on the primary table.

If that is not possible, there are several approaches for limiting duplicates created because of a join:

- Rewrite to use the `subselect` method approach. For reverse inner joins, the query optimizer often performs better with `subselect` than using `join` or `outerJoin`. See "Using Subselect for Reverse Inner Joins" on page 152 for details and examples. However, every situation is different and in some cases using the `join` method might perform better. For example, if the secondary table including predicates are not very selective. In other words, if the secondary clause returns lots of results, consider using `join`. For important queries, Guidewire recommends trying both approaches under performance testing. If you use `join` for a reverse join, use the two-parameter method signature including the table name and column name in the joined table.
- Call the `withDistinct` method on `Query` to limit the results to distinct results from the join. Pass the value `true` as an argument to limit the query to distinct results. To turn off this behavior later, call the method again and pass `false` as a parameter. For example:

  ```
  var q = Query.make(SampleParent)
  q.withDistinct(true)
  q.join("MyChildObjectProperty")
  ```
- If you add predicates to the subquery, use the `having` method. Suppose you add predicates to a subquery to limit what your query matches. For example, only matching entities on the primary table if a child object has a certain property match a specific value.

  The following example adds predicates after the join using properties on the joined table:

  ```
  var q = Query.make(SampleParent)
  q.join("MyChildObjectProperty")
  q.compare("ChildPriority", Equals, 5)
  ```
  When you add predicates after the join, you can control whether the predicates added after the join create duplicates on the primary table using another approach. To do this, call the `having` method with no arguments. Next, add your predicates to the result of the `having` method. For example:

  ```
  var q = Query.make(SampleParent)
  q.join("MyChildObjectProperty")
  q.having().compare("ChildPriority", Equals, 5)
  ```
  The addition of the `having` method adds a SQL `HAVING` keyword. This tells the database that you are interested in determining the set of items on the primary table, not with each match on the secondary table. The result returns no additional duplicates on the primary table (the original query). As mentioned earlier, if you can rewrite to use the `subselect` method approach instead of using the `join` method, the query optimizer typically performs better. See "Using Subselect for Reverse Inner Joins" on page 152 for details and examples.

  If you do not add predicates to the join table, you cannot use this approach.

Adding Predicates to Join Queries

You can add comparison predicates on the primary table by adding to the original query you created.

For example:

```
var q = Query.make(SampleParent)

q.compareIn("ID", myEntityIDs)
q.join("MyChildObjectProperty")
```

However, you can also add predicates that use properties on the joined table, by calling the predicate methods on the joined table. The joined table is the result of the join method.

Thus, you can either save the result of the `join` method and add the predicates:

```
var q = Query.make(SampleParent)
var childTable = q.join("MyChildObjectProperty")
childTable.compare("ChildPriority", Equals, 5)
```

Alternatively, link them in line as a builder pattern:

```
var q = Query.make(SampleParent)
q.join("MyChildObjectProperty").compare("ChildPriority", Equals, 5)
```

You can also add additional predicates to the primary table even after creating the join:

```
var q = Query.make(SampleParent)

var childTable = q.join("MyChildObjectProperty")

// add predicates to join table -- uses namespace of joined table table for properties
childTable.compare("ChildPriority", Equals, 5)

// add predicates to original query -- uses namespace of original table for properties
q.compareIn("ID", myEntityIDs)
```

For reverse joins, if you add predicates to the join table and you want maximum one entity on the primary table for each match, another API can help. For more information about the `having` method, see later in this section (“Duplicates with Reverse Joins” on page 155).

Allowing Queries to Find Retired Entities

By default, queries do not find retired entities. Retired entities are logically like deleted entities but remain in the database with a `Retired` flag set to true. Not all entities are retirable. This quality is defined in the data model configuration files. To find retired entities also, set the query’s `FindRetired` property to `true`:

```
var q = Query.make(SampleParent)
q.compare("ParentProperty1", Equals, 10)
q.FindRetired = true
```

Advanced Inline Views

You can optionally create what SQL calls an inline view of these extra columns on the join table. This is an advanced feature which is not normally needed in real-world code. Use this feature only if necessary due to the performance implications. If you have questions about this, contact Guidewire Customer Support.

Inline views let you add extra columns to the outer query. In other words, you can add predicates to the outer query that reference those columns.

To create an inline view, call the `inlineView` method on the query. It returns a query with the new inline view and automatically selects all referenced columns from that query in the `select` statement. It takes the following arguments:

- `joinColumnOnThisTable` - the name of the join column (`String`)
- `inlineViewQuery` - the query (`Query`)
- `joinColumnOnViewTable` - the name of the join column on the view table (`String`)

The method returns a new query with the inline view. Technically the return type of the method is a `Table` object.

For example, suppose you create two queries, a view query and an outer query:

```
var viewQuery = Query.make(TestA)
var outerQuery = Query.make(TestE)
```

Next, make an inline view to add the `E` column:

```
var inlineView = outerQuery.inlineView("ID", viewQuery, "E")
```

Next, use the `E` column from new predicates on the **outer** query:

```
outerQuery.compare("E", GreaterThan, inlineView.getColumnRef(DBFunction.Max("A")))
```

Test the code:

```
print(outerQuery.select().AtMostOneRow.ID)
```

This prints the ID of a `TestE` entity where `TestE.E` is greater than the maximum of all values of `A` in all `TestA` instances related to this `TestE`. Thus generates the SQL

```
select * from TestE INNER JOIN (select E, Max(A) MAX_A from TestA) testA_view
  GROUP BY E ON TestE.ID = testA_view.E AND TestE.E > testA_view.MAX_A
```

Notice that automatically predicates on the join table can use columns on the join table or the primary table.

Additional Join Examples

Inner Join and Constant Database Functions

Suppose you want to find:

- entities of type `SampleParent`
- only entities that have child subobjects of type `SampleChild`, which have a `Parent` foreign key column.
- the `Sample.Priority` property has a value between the minimum of the `SampleChild.Priority` values and the maximum of the `SampleChild.Priority` properties. (To perform the minimum and maximum commands, use the `Min` and `Max` static methods off of the `DBFunction` object, as shown in the following example.)

This type of query requires a join between multiple tables. The following example creates this query

```
var q = Query.make(SampleParent)
q.compareIn("ID", myEntityIDs)
var childTable = q.join(SampleChild, "parent")

q.having().between("ParentPriority", DBFunction.Min(childTable.getColumnRef("ChildPriority")),
  DBFunction.Max(childTable.getColumnRef("ChildPriority")))
```

Example Foreign Key and Retirable

This query searches for `User` entities with its `ExternalUser` property `false` and with a `Contact` associated with them:

```
var q = Query.make(User)

q.compare("ExternalUser", Equals, false)
q.join("Contact")
```

This produces the SQL

```
[false]
SELECT  FROM cc_user qRoot INNER JOIN (cc_contact cc_contact_1) ON cc_contact_1.ID=qRoot.ContactID
AND cc_contact_1.Subtype = 13 AND cc_contact_1.Retired = 0  WHERE qRoot.ExternalUser = ? AND
qRoot.Retired = 0
```

Notice that Gosu automatically adds the check for the `Retired` property, since the default is to only find non-retired entities. (See “Allowing Queries to Find Retired Entities” on page 157.)

chapter 8

Find Expressions

A Gosu `find()` statement lets you query the database and then iterate the results. This topic describes important concepts in writing `find()` statements to retrieve required data from the database.

> **IMPORTANT** Although `find` expressions are supported, there is a newer improved system called query builder APIs. For new code, Guidewire **strongly recommends** that you use the newer query builder APIs rather than `find` expressions. For more information, see “Query Builder” on page 129.

This topic includes:

Basic Find Expressions

Gosu `find` expressions lets you query the database using the existing product data model, including data model extensions. You can write very useful queries in a comprehensive yet concise manner.

The main `find` expression uses the following syntax:

```
var q = find ( [var] <identifier> in <query-path-expression> where <where-clause-expression> )
```

Finder expressions have the following parts:

- **The identifier.** This symbol labels the table (or join table) in the finder so you can more easily express criteria.
- **The ‘in’ clause as a query path expression.** The `in` clause query path looks similar to a Gosu object path but `in` clause query path begins with an entity type literal (or subtype literal) at its root. In contrast, an object path begins with a reference to an entity instance.

  A query path expression: `Claim.Claimnumber`

  An object path expression: `claim.claimnumber`

- **The 'where' clause expression.** This part defines the criteria for the finder. It uses all the same logical and relational expressions used elsewhere in Gosu. For instance, `or` and `and` logical expressions follow the same syntax rules, as do all the relational expressions: `==`, `!=`, `>`, `<`, `>=`, and `<=`. As everywhere else in Gosu, you may combine and nest expressions any way you like.

Note the query path expression defined in the `in` clause must be defined in terms of an *absolute* object path with an entity type (or subtype) literal at its root. Conversely, a query expression in a `where` clause must be defined in terms of a *relative* object path with the finder's identifier at the root. This compile-time restriction improves finder readability and consistency, and also prevents overly complex queries from entering the system.

Evaluating a `find()` expression results in a `Query` instance, which you can use to iterate across the results. For more information about using results of a find query (`Query` instances), see "Using the Results of Find Expressions (Using Query Objects)" on page 164.

Express subexpressions as you would in any other Gosu construct. However, `where` clause expressions can only test a value on a property that is a **direct** property of the finder's query path identifier. For example, if you are iterating across `Claim` entities, a simple query can only get properties directly from `Claim`, not properties of subobjects referenced by the claim.

> **IMPORTANT** If you need `where` clause expressions that traverse more deeply in a path expression than one level (direct properties), use an `exists` expression in your `find` query. The `exists` expressions are incredibly powerful for you to execute complex application logic. Exists expressions can generate database `join` commands in the generated SQL query. See "Using Exists Expressions in Finders for Database Joins" on page 162 for more information.

Example 1

The following simple `find` expression queries for all claims with a `LossType` of `"AUTO"`.

```
var q = find( claim in Claim where claim.LossType == "AUTO" )
```

This expression declares the variable `claim` and assigns it the type specified in the query path expression, which in this case is `Claim`. The `var` keyword before the variable `claim` is optional. The `where` clause expression in this example defines expressions in terms of `claim`. In this simple example, `claim.LossType == "AUTO"` matches all claims with `"AUTO"` loss types.

Example 2

In the following more complicated example, an expression defines a more complex query path expression. The expression joins `Claim` to `Policy`, then queries for all claims with a policy number of "54-123456".

```
var q = find( var policy in Claim.Policy where policy.PolicyNumber == "54-123456" )
```

Note the query path expression in the `in` clause is an *absolute* object path with an entity type (or subtype) literal at its root. The query expression in the `where` clause is a *relative* object path with the finder's identifier at the root.

Example 3

In the following example, the expression uses a subtype in the `in` clause to restrict the query to Person-type contact entities (as opposed to Company, for example). The query searches for males born after January 1, 1982.

```
var q = find(var people in Person where people.Gender == "M" and people.DateOfBirth >= "1982-01-01")
```

Find Expressions Using AND/OR

Logical `AND` or `OR` expressions in a `where` clause are fundamentally the same as those used elsewhere in Gosu.

The following example expands the code shown in Example 1 in the previous section to refine the criteria. The query now excludes claims with accident type 46 (*collision with fixed object*).

```
var q = find( c in Claim where c.LossType == "AUTO" and c.AccidentType != "46" )
```

It adds the “`and`” operator to refine the query with the additional `c.AccidentType != "46"` expression.

The following example expands Example 1 again and refines the query even further to include all claims with unassigned loss types.

```
var q = find( c in Claim where c.LossType == null or c.LossType == "AUTO" and c.AccidentType != "46" )
```

This construction adds the `c.LossType == null` criteria separated by with an `or` operator. Appending this condition would produce the same result as the `and` operator has precedence. To change the criteria so that Gosu evaluates the `or` expression first, use parenthesis to change the order of evaluation.

```
var q = find( c in Claim where (c.LossType == null or c.LossType == "AUTO")
        and c.AccidentType != "46" )
```

Since parentheses surround the two parts of the `or` expression, Gosu evaluates this condition differently then if the parentheses were not there. You may nest expressions in this manner to any level necessary.

Find Expressions Using Equality and Relational Operators

Gosu uses the following equality and relational operators in logical expressions:

Equality expressions
`==`
`!=`

Relational expressions
`<`
`>`
`<=`
`>=`

For more information about these expressions, see “Equality Expressions” on page 66 and “Relational Expressions” on page 73.

Like logical `and` and logical `or` expressions, equality and relational expressions in a `where` clause are fundamentally the same as those used elsewhere in Gosu.

However, there are two restrictions that differentiate these expressions used in a `where` clause:

- **Left-hand side must be a relative query path expression.** The left-hand side of the operator must be a relative query path expression with the finder’s identifier as the root. In addition the query path must only be one property deep, which means only the identifier’s immediate properties can be referenced. For instance, if the query started with “`find c in Claim where` ...”, then the expression `c.AssignedGroup` is legal. However, `c.AssignedGroup.GroupType` is illegal because the property `GroupType` is not a direct property of `Claim`. The previous examples all demonstrate this rule.

  IMPORTANT If you need `where` clause expressions that traverse more deeply in a path expression than one level (direct properties), use an `exists` expression in your `find` query. The `exists` expressions are incredibly powerful tools for you to execute complex application logic. Such expressions can generate database join commands in the generated SQL query. See “Using Exists Expressions in Finders for Database Joins” on page 162 for more information.

- **Right-hand side must not include a query path expression.** The right-hand side of the operator must not reference a query path expression. This means that queries cannot compare query properties dynamically. All other Gosu expressions are legal on the right-hand side.

  There are some subtle distinctions, however:

- The expression `Claim.LossDate == Exposure.AssignmentDate` is *illegal* because `Exposure.AssignmentDate` is a query path expression and not an object path expression.
- The expression `Claim.LossDate == exposure.AssignmentDate` is *legal*, assuming that `exposure` is an instance of an exposure (an instance of the `Exposure` entity type).

The precedence rules for equality and relational expressions if used in a `where` clause are the same as within other Gosu constructs. This means that relational operators always have precedence over equality operators.

Because a find query is a high-level abstraction for generating database SQL queries, virtual properties or method calls are **illegal** in the `where` clause of a `find` expression. A property is *virtual* if Gosu generates it dynamically by Gosu at the time it is accessed. Virtual properties do not directly mirror a single actual database column so there is typically no direct translation of what the corresponding SQL would be. If you are not sure whether a property is virtual, refer to the application Data Dictionary.

For related information, see “Properties” on page 172.

IMPORTANT You cannot reference an entity’s virtual properties or methods in a `find` query.

Find Expressions Using ‘Where…In’ Clauses

To test if the value on the left hand side is present in an array, use an additional `in` clause from within the `where` clause.

For instance, the following example finds claims with a `LossCause` matching one or more literal values in a dynamically created array.

```
find( c in Claim where c.LossCause in new LossCause[] {"vandalism","theftparts","theftentire"} )
```

This next example finds all the activities directly related to a specified claim’s matters.

```
find( a in Activity where a.Matter in claim.Matters )
```

As with equality and relational operators, the `in` operator requires a relative query expression on the left-hand side. In addition, the right hand side must be coercible to an array that is compatible with the left-hand side’s type. Consequently, the legal right hand side values include arrays, collections, result sets, and `String` objects (strings are treated as arrays of characters).

Using Exists Expressions in Finders for Database Joins

To create search criteria that expand into a finder’s query-path-expression deeper than one level (direct properties), use the `exists` construction within your `where` expressions. In essence, an `exists` expression creates a sort of subfinder using a containing finder’s related property. Using `exists` expressions are an incredibly powerful and useful method for you to execute complex database queries. An `exists` expression is especially useful if you need to test a value on a property that is not a **direct** property of the finder’s query path identifier. In other words, if a simple query iterating across a `Claim` entities can test properties directly on `Claim`, not subproperties of objects that are referenced by the claim.

An `exists` expression is a Gosu feature not strictly tied to find expressions or database queries. However, `exists` expressions are particularly useful within a `find` query. See “Existence Testing Expressions” on page 68 for other uses of an `exists` expression.

The syntax of an exists clause in a where expression is:

```
where ... exists ( subQueryIdentifier in queryPathExpression where whereExpression )
```

The *`queryPathExpression`* must be a query path expression with an entity type literal (or entity subtype literal) at the root. See “Basic Find Expressions” on page 159 for more information.

The *`whereExpression`* must be a Gosu standard `where` clause. See “Basic Find Expressions” on page 159 for more information.

Note the query path expression defined in the `in` clause must be defined in terms of an *absolute* object path with an entity type (or subtype) literal at its root. Conversely, a query expression in a `where` clause must be defined in terms of a *relative* object path with the finder's identifier at the root.

For example, the following is an example finder expression:

```
find( c in Claim where c.LossType == "AUTO" and exists(
  group in Claim.Exposures.AssignedGroup where group.GroupType == "autofasttrack" ) )
```

If an `exists` expression appears within a `find` expression, Gosu optimizes the database query to retrieve data based on potentially complex Gosu expressions. Gosu might generate complex database queries including database join commands if needed. Database joins are a feature of databases and query languages that push resource-intensive work and intelligence to the database if querying data from two or more database tables.

Generally speaking you do not need to worry about what SQL is generated, including any hidden database join operations. However, some finder expressions may have large performance considerations. Guidewire recommends you consider both frequency and context of a query's execution and assess the performance implications. Always run performance tests with large data sets and check for unexpected performance issues, experimenting with variants of your query expressions as necessary.

IMPORTANT Although all find expressions generate SQL queries to the database from Gosu expressions, `find` expressions do not directly support full SQL syntax. For example, you cannot directly use SQL `JOIN` command because that keyword is not supported.

Example

The following example for ClaimCenter queries for all activities with a particular activity pattern for a particular claim number:

```
var activities = find (activity in activity
  where activity.ActivityPattern == ActivityPattern( "default_data:10003" /* contact_insured */ )
  and exists (c in activity.Claim where c.ClaimNumber == "235-53-365889"))
```

Fixing Invalid Queries by Adding Exists Clauses

For some types of queries, you must use an `exists` expression to properly convey the application logic in an efficient way. Some types of queries generates Gosu syntax errors if you break the rules of a basic query, and in some cases it can be rewritten simply using exists expressions.

For example, the following query expression is **illegal** because the `GroupType` property is not a direct property of the entity for which the query searches. The query attempts to find all claims in which the loss type is `"AUTO"` and which has an exposure with assigned group of type `autofasttrack`.

```
//Illegal expression
find( c in Claim where c.LossType == "AUTO" and c.Exposures.AssignedGroup.GroupType == "autofasttrack")
```

The path expression `c.Exposures.AssignedGroup.GroupType` is illegal in a simple query because `GroupType` is not a direct property of a `Claim`, which is the type of the iteration variable `c`. If a finder's criteria must traverse deeper into the primary query path to retrieve a property, the query must define an `exists` expression. The `exists` expression must describe the expanding part of the criteria beyond the original entity.

The following valid example correctly finds all claims in which the loss type is `"AUTO"` and which has an exposure with assigned group of type `autofasttrack`.

```
find( c in Claim where c.LossType == "AUTO" and exists(
  group in Claim.Exposures.AssignedGroup where group.GroupType == "autofasttrack" ) )
```

Note: For those people familiar with implementation details of SQL joins, the example generates a reverse join from claim to exposure and a forward join from exposure to group. The reverse join is actually a subquery nested in the outer finder's query.

Combining Exists Expressions

A finder may nest and combine `exists` expressions to any level desired. In other words, `exists` expressions can contain and combine one or more other `exists` expressions, and the outer find query can contain more than one `exists` expression.

Find Expressions Using Special Substring Keywords

You can also use `find` expressions to search a portion of a string value, in other words to search for a substring. To search on a few initial characters, use the special keyword `startswith`.

For example, the following query uses `startswith`:

```
var query = find (var branch in Group where branch.Name startswith "Ac")
```

If a branch's name starts with `Ac`, it succeeds. For example, the name `"Acme Insurance"`.

To find a substring embedded in a larger string, use the special keyword `contains`. For example:

```
var query = find (clm in Claim where clm.ClaimNumber contains "12")
```

Using the Results of Find Expressions (Using Query Objects)

Evaluating a `find` expression results in a `Query` object, which you use to navigate or otherwise access items in the result. Items in `Query` objects are entity instances of the type declared in the query path expression. For instance, if you are querying `Contact` objects, the resulting type is a `ContactQuery`. Query types have the same inheritance hierarchy that the entities have, for example: `PersonQuery` extends `ContactQuery`.

You extract the results of a query by using the `query.iterator` method, which returns an `Iterator` to use to navigate within the results. You can avoid an explicit call to the `iterator` method by using the `Query` object in a `for...in` statement. The `iterator` method is used indirectly whenever you use the query in a `for...in` statement.

When Data for Find Expressions Is Loaded

Defining a `find` expression to obtain a query object does not immediately load data from the database. The server loads data from the database into a find query only at the time your code uses the query. For example, the server loads data at the time your code gets an item from the iterator or calls the `query.getCount` method.

Performance Considerations for Find Expressions

If you execute multiple `find` expressions, each expression sets up a separate query against the database. Write your code in such a way as to minimize the number of queries that run against the database.

IMPORTANT Be careful not to use too many criteria in `find` expressions. Be careful to design your `find` expressions to extract only the data you need.

Setting the Page Size for Retrieving Results from Find Expressions

The `Iterator` object returned by the `query.iterator` method retrieves data from the server all at once or one page at a time. Page-based retrieval of query data can help avoid large result sets from loading into memory all at once. Loading large results into memory all at once can lead to performance problems or out-of-memory errors.

You turn on page-level retrieval and limit the page size for an individual `Query` object by calling the `setPageSize` method. Specify the number of items per page. If you do not set a page size explicitly on a `Query` object, results load at one time without page-based retrieval.

Basic Iterator Example

The following example demonstrates how to navigate the results of a simple `find()` expression using the `iterator()` method. Notice how the `for...in` expression can directly handle an iterator. This is the preferred way to navigate a `Query`.

```
var query = find( p in Claim.Policy where p.PolicyNumber == "54-123456" )
for( claim in query ) {
  print( claim.ClaimNumber )
}
```

Large Query Finder Results Example

If there are concerns about producing an excessively large `Query`, Guidewire recommends that you maintain an entity count and terminate the iteration, if necessary.

The following example throws an exception indicating that the results are too large to process.

```
var query = find( p in Claim.Policy where p.PolicyNumber == "54-123456" )
var iCount = 0
for( claim in query ) {
  print( claim.ClaimNumber )
  iCount = iCount + 1
  if( iCount == 1000 ) {
    throw "Too many claims to process."
  }
}
```

Sort Results Example

Finally, you can set the sort order in the `Query` using the following methods:

- `addAscendingSortColumn()`
- `addDescendingSortColumn()`

The following example sorts claims by *`ClaimNumber`*.

```
var query = find( p in Claim.Policy where p.PolicyNumber == "54-123456" )
query.addAscendingSortColumn( "Claim.ClaimNumber" )
for( claim in query ) {
  print( claim.ClaimNumber )
}
```

Returning a Single Row of Finder Results

You can use the `getAtMostOneRow()` method to retrieve a single item from the `find()` expression.

- If a single item exists, the method returns it.
- If no item exists, the method returns `null`.
- If more than one item exists, the method throws an exception.

Use this method only if you are sure that `find()` expression returns a single item. Otherwise, you need to handle the exception condition that occurs.

For example, the following code calls the Gosu class function `findActivityPattern()` to find a matching activity pattern and prints an error if the function call returns more than one item.

```
try {
  var pattern = new act.findActivityPattern( "initial_30day_review", "general")
} catch (e) {
  print("Query returned more than one row.")
}

//Gosu Class Function
package act

class findActivityPattern {
  public function findActivityPattern( code : String, type : ActivityType ) : ActivityPattern {
    var query = find( ap in ActivityPattern where ap.code == code and ap.Type == type)
    return query.getAtMostOneRow()
```

```
  }
}
```

See "“Classes”, on page 169" for details on how to write Gosu classes. Also, see “Exception Handling” on page 93 for more information on using `try...catch...` statements.

Found Entities Are Read-only Until Added to a Bundle

Entities that you iterate across after a find query are read-only by default. The entity is loaded in a read-only *bundle*, which is a collection of entities loaded from the database into server memory. By default, find query results are returned in their own read-only bundle separate from the active read-write bundle of any running code.

To change a read-only entity’s properties, you must move the entity to a new writable bundle. Typical code adds the entity to the *current bundle* of the running code. For example, if your Gosu code is triggered from a rule set or a plugin method, there is a current bundle that you can access. Simply add each entity to the new bundle to permit changes to the entity and mark the entity as changed. After the entity changes and the new bundle commits successfully, all entity changes are copied to the database. Remember that an entity cannot safely exist in more than one writable (read-write) bundle.

To move the entity to a writable bundle, call the `add` method on a bundle and save the result of the `add` method.

IMPORTANT If you passed a read-only entity to a writable bundle, the return result of the `add` method is a cloned instance of the entity you passed to the method. If you want a reference to the entity so you can make further changes, you must keep a reference to the **return result** of the `add` method. Do not modify the original entity reference. Do not keep a reference to the original entity.

For example:

```
// get the current (ambient) bundle of the runing code
var bundle = gw.transaction.Transaction.getCurrent()

var query = find( p in Claim.Policy where p.PolicyNumber == "54-123456" )

for( claim in query ) {
  var c = bundle.add(claim)

  // make changes to one or more properties directly on the entity or its subobjects
  c.MyProperty1 = true
}
```

IMPORTANT For more information about bundles and transactions, see “Bundles and Transactions” on page 275, which discusses the issue and related APIs in much more detail.

Queries Are Always Dynamic

A query is always dynamic and returns results that may change if you use the object again. Some results may have been added, changed, or removed from the database from one use of the query object to another use of the query, even within the same function.

A common mistake is calling the `count` method of a query to get the current count of items and then rely on that number remaining constant. Although that number might be useful in some contexts, it might be different from the number of items returned from a query even if you iterate across it immediately afterward. Some results may add, change, or remove from the database between the time you call the `count` method and the time you iterate across it.

Bad Example:

```
// Bad example. Do not do this...

Var myResult = find ( /* some query here... */ )
```

```
Var myArray = new Claim[ myResult.count() ];

For ( x in myResult index y )
{
  myArray[y] = x;
}
```

In most cases, you severely risk of an “array out of bounds” error at run time.

Instead, iterate across the set and count upward, appending items to a `List` as necessary.

chapter 9

Classes

Gosu classes encapsulate data and code for a specific purpose. You can subclass and extend existing classes. You can store and access data and functions (also called methods if part of a class) on an instance of the class or on the class itself.

For details on using Guidewire Studio to create and manage Gosu classes, see the “Gosu Classes” on page 107 in the *Configuration Guide*.

Gosu classes are the foundation for syntax of syntax for interfaces, enumerations, and enhancements. Some of the information in this topic applies to those features as well. For example, the syntax of variables, methods, and modifiers are the same in interfaces, enumerations, and enhancements.

Related topics:

- “Interfaces”, on page 191
- “Enumerations”, on page 189
- “Enhancements”, on page 209

This topic includes:

What Are Classes?

Gosu classes encapsulate data and code to perform a specific task. Typical use of a Gosu class is to write a Gosu class to encapsulate a set of Gosu functions and a set of properties to store within each class *instance*. A class instance is a new in-memory copy of the object of that class. If some Gosu code creates a new instance of the class, Gosu creates the instance in memory with the type matching the class you instantiated. You can manipulate each object instance by getting or setting properties. You can also trigger the class's Gosu functions. If functions are defined in a class, the functions are also called *methods*.

You can also extend an existing class, which means to make a subclass of the class with new methods or properties or different behaviors than existing implementations in the superclass.

Gosu classes are analogous to Java classes in that they have a package structure that defines the namespace of that class within a larger set of names. For example, if your company is called Smith Company and you were writing utility classes to manipulate addresses, you might create a new class called `NotifyUtils` in the namespace `smithco.utilities`. The fully-qualified name of the class would be `smithco.utilities.NotifyUtils`.

You can write your own custom classes and call these classes from within Gosu, or call existing Guidewire-written classes. You create and reference Gosu classes by name just as you would in Java. For example, suppose you define a class called `Notification` in package `smithco.utilities` with a method (function) called `getName()`.

You can create an instance of the class and then call a method like this:

```
// create an instance of the class
var myInstance = new smithco.utilities.Notification()

// call methods on the instance
var name = myInstance.getName()
```

If desired, you can also define data and methods that belong to the class itself, rather than an instance of the class. This is useful for instance to define a library of functions of similar purpose. The class encapsulates the functions but you never need to create an instance of the class. You can create static methods on a class independent of whether any code ever creates an instance of the class. You are not forced to choose between the two design styles. For more information, see "Static Modifier" on page 183.

For details on using Studio to create and manage your Gosu classes, see "ClaimCenter Studio and Gosu" on page 105 in the *Configuration Guide*.

If desired, you can write Gosu classes that extend from Java classes. Your class can include Gosu generics features that reference or extend Java classes or subtypes of Java classes. See "Gosu Generics", on page 221 for more information about generics.

Creating and Instantiating Classes

In Studio, create a class by right-clicking on a package name under the **Classes** part of the resource pane. Studio creates a simple class upon which you can build. Studio creates the package name, class definition, and class constructor. You can add class variables, properties, and functions to the class. Within a class, functions are also called *methods*.

After creating a new class, add additional class variables, properties, and functions to the class.

> **Note:** Within a class, functions are also called *methods*. This is standard object-oriented terminology. This documentation refers to functions as methods in contexts in which the functions are part of classes.

If you create a new class, the editor creates a template for a class upon which you can build. The editor creates the package name, class definition, and class constructor. You can add class variables, properties, and functions to the class. Within a class, functions are also called *methods*.

Add variables to a class with the `var` keyword:

```
var myStringInstanceVariable : String
```

You can optionally initialize the variable:

```
var myStringInstanceVariable = "Butter"
```

Define methods with the keyword `function` followed by the method name and the argument list in parentheses, or an empty argument list if there are no arguments to the method. The parameter list is a list of arguments, separated by commas, and of the format:

```
parameterName : typeName
```

For example, the following is a simple method:

```
function doAction(arg1 : String)
```

A simple Gosu class with one instance variable and one public method looks like the following:

```
class MyClass
{
  var myStringInstanceVariable : String

  public function doAction(arg1 : String)
  {
    print("Someone just called the doAction method with arg " + arg1)
  }
}
```

Constructors

A Gosu class can have a *constructor*, which is like a special method within the class that Gosu calls after creating an instance of that type. For example, if Gosu uses code like "`new MyClass()`", Gosu calls the `MyClass` class's constructor for initialization or other actions. To create a constructor, name the method simply `construct`. For example:

```
class Tree
{
  construct()
  {
    print("A Tree object was just created!")
  }
}
```

If desired, you can delete the class constructor if you do not need it.

Your class might extend another class. If so, it is typically appropriate for your constructor to call its superclass constructor. To do this, use the super keyword. It must be the first line in the subclass constructor. For example

```
class Tree extends Plant
{
  construct()
  {
    super()
    print("A Tree object was just created!")
  }
}
```

If you call `super()`, Gosu calls the superclass no-argument constructor. If you call `super(`*`parameter_list`*`)`, Gosu calls the superclass constructor that matches the matching parameter list. Note that you can call a superclass constructor with different number of arguments or different types than the current constructor.

Static Methods and Variables

If you want to call the method directly on the class itself rather than an instance of the class, you can do this. This feature is called creating a *static method*. Add the keyword `static` before functions that you declare to make them static methods. For example, instead of writing:

```
public function doAction(arg1 : String)
```

Instead use this:

```
static public function doAction(arg1 : String)
```

For more information, see "Static Modifier" on page 183.

Although public variables are a supported part of the language for compatibility with other languages, Guidewire strongly recommends public properties backed by private variables instead of using public variables.

In other words, in your new Gosu classes do this:

```
private var _firstName : String as FirstName
```

Do not do this:

```
public var FirstName : String
```

For more information about defining properties, see “Properties” on page 172.

IMPORTANT Guidewire strongly recommends public properties backed by private variables instead of using public variables. Do not use public variables in new Gosu classes. See “Properties” on page 172 for more information.

Creating a New Instance of a Class

Typically you want to create an instance of a class. Each instance (in-memory copy) has its own set of data associated with it. The process of constructing a new in-memory instance is called *instantiating a class*. To instantiate a class, use the `new` operator:

```
var e = new smithco.messaging.QueueUtils()
```

You can also use object initializers allow you to set properties on an object immediately after a `new` expression. Use object initializers for compact and clear object declarations. They are especially useful if combined with data structure syntax and nested objects. A simple version looks like the following:

```
var sampleClaim = new Claim(){ :ClaimId = "TestID" }
```

For more information on `new` expressions and object initializers, see “New Object Expressions” on page 70.

Note: You can use Gosu classes without creating a new instance of the class using static methods, static variables, and static properties. For more information, see “Static Modifier” on page 183.

Naming Conventions for Packages and Classes

The package name is the namespace for the class, interface, enhancement, enumeration, or other type. Defining a package prevents ambiguity about what class is accessed.

Package names must consist completely of lowercase characters. To access classes or other types in another package namespace, see “Importing Types and Package Namespaces” on page 75. Class names or other type names must always start with an initial capital letter. However, the names may contain additional capital letters later in the name for clarity.

Guidewire recommends the following package naming conventions:

Type of class	Package	Example of fully qualified class name
Classes you define	*customername.subpackage*	smithco.messaging.QueueUtils
Guidewire Professional Services classes	gwservices.*subpackage*	gwservices.messaging.QueueUtils

WARNING Classes and enumerations must never use a package name with the prefix “`com.guidewire.`” or the prefix “`gw.`” Those package namespaces are for Guidewire internal classes only. For more information about reserved packages, see “Importing Types and Package Namespaces” on page 75. If you write Gosu enhancements, the naming rules are slightly different (see “Enhancement Naming and Package Conventions” on page 212).

Properties

Gosu classes can define properties, which appear to other objects like variables on the class in that they can use simple intuitive syntax with the period symbol (`.`) to access a property for setting or getting the property.

However, you can implement get and set functionality with Gosu code. Although code that gets or sets properties might simply get or set an instance variable, you can implement properties in other more dynamic ways.

To get and set properties from an object with `Field1` and `Field2` properties, just use the period symbol like getting and setting standard variables:

```
// create a new class instance
var a = new MyClass()

// set a property
a.Field1 = 5

// get a property
print (a.Field2)
```

In its most straightforward form, a class defines properties like functions except with the keywords "`property get`" or "`property set`" before it instead of "`function`". The `get` property function must take zero parameters and the `set` property function always takes exactly one parameter.

For example, the following code defines a property that supports both set and get functionality:

```
class MyClass {
  property get Field3() : String   {
    return "myFirstClass" // in this simple example, do not really return a saved value
  }
  property set Field3(str : String)   {
    print (str) // print only ---- in this simple example, do not save the value
  }

}
```

The `set property` function does not save the value in that simple example. In a more typical case, you probably want to create a class instance variable to store the value in a private variable:

```
class MyClass {
  private var _field4 : String

  property get Field4() : String   {
    return _field4
  }
  property set Field4(str : String)   {
    _field4 = str
  }

}
```

Although the data is stored in private variable `_field4`, code that accesses this data does not access the private instance variable directly. Any code that wants to use it simply uses the period symbol (`.`) with the property name:

```
var f = new MyClass()
f.Field4 = "Yes" // sets to "Yes" by calling the set property function
var g = f.Field4 // calls the get property function
```

For some classes, your property getter and setter methods may do very complex calculations or store the data in some other way than as a class variable. However, it is also common to simply get or set a property with data stored as a common instance variable. Gosu provides a shortcut to implement properties as instance variables using *variable alias* syntax using the `as` keyword followed by the property name to access the property. Guidewire recommends this approach to make simple automatic getter and setter property methods backed by an class instance variable.

For example, the following code is functionally identical to the previous example but is much more concise:

```
class MyClass {
  private var _field4 : String as Field4
}
```

Guidewire strongly recommends public properties backed by private variables instead of using public variables.

In other words, write your Gosu classes to look like:

```
private var _firstName : String as FirstName
```

Do not write your classes to look like:

```
public var FirstName : String
```

IMPORTANT Guidewire strongly recommends public properties backed by private variables instead of using public variables. Do not use public variables in new Gosu classes.

Code defined in that class does not need to access the property name. Classes can access their own private variables. In the previous example, other methods in that class could reference `_field4` or `_firstname` variables rather than relying on the property accessors `Field4` or `FirstName`.

Read Only Properties

The default for properties is read-write, but you can make a property read-only by adding the keyword `readonly` before the property name:

```
class MyClass {
  private var _firstname : String as readonly FirstName
}
```

Properties are Actually Virtual Like Functions

In contrast to standard instance variables, `get property` and `set property` functions are *virtual*, which means you can override them in subclasses and implement them from interfaces. The following illustrates how you would override a property in a subclass and you can even call the superclass's get or set property function:

```
class MyClass
{
  var _easy : String as Easy
}

class MySubClass extends MyClass
{
  override property get Easy() : String
  {
    return super.Easy + " from MySubClass"
  }
}
```

The overridden property get function first calls the implicitly defined get function from the superclass, which gets class variable called `_easy`, then appends a string. This get function does **not** change the value of the class variable `_easy`, but code that accesses the `Easy` property from the subclass gets a different value.

For example, if you write the following code in the Gosu Tester:

```
var f = new MyClass()
var b = new MySubClass()

f.Easy = "MyPropValue"
b.Easy = "MyPropValue"

print(f.Easy)
print(b.Easy)
```

This code prints:

```
MyPropValue
MyPropValue from MySubClass
```

Property Paths are Null Tolerant

One notable difference between Gosu and some other languages is that property accessor paths in Gosu are *null tolerant*, also called *null safe*. This affects only expressions separated by period characters that access a series of *instance variables* or *properties*. In other words, the form:

```
obj.Property1.Property2.Property3
```

In most cases, if any object to the left of the period character is `null`, Gosu does not throw a *null pointer exception* (NPE) and the expression returns `null`. Gosu null-safe property paths tends to simplify real-world code.

Often, a `null` expression result has the same meaning whether the final property access is `null` or whether earlier parts of the path are `null`. For such cases in Gosu, do not bother to check for `null` value at every level of the path. This makes your Gosu code easier to read and understand.

For example, suppose you had a variable called `house`, which contained a property called `Walls`, and that object had a property called `Windows`. The syntax to get the `Windows` value is:

```
house.Walls.Windows
```

In some languages, you must worry that if `house` is `null` or `house.Walls` is `null`, your code throws a `null` pointer exception. This causes programmers to use the following common coding pattern:

```
// initialize to null
var x : ArrayList<Windows> = null

// check earlier parts of the path for null to avoid a null pointer exceptions (NPEs)
if( house != null and house.Walls != null ) {
   x = house.Walls.Windows
}
```

In Gosu, if earlier parts of a pure property path are `null`, the expression is valid and returns `null`. In other words, the following Gosu code is equivalent to the previous example and a null pointer exception never occurs:

```
var x = house.Walls.Windows
```

However, method calls are not null safe. This means that if the right side of a period character is a method call, Gosu throws a null pointer exception if the left side of the period is `null`.

For example, suppose you have a method access followed by a property/variable access:

```
myaction().Walls
```

If the `myaction` method returns `null`, the property access that follows is null safe, and simply returns `null`. Because Gosu assumes property and variable access does not have side effects, it is safe to skip the property access and just return `null`.

In contrast, a **method call** throws an exception if the method's target (the left side of the period character) is `null`. For example, consider this expression:

```
house.Walls.myaction()
```

If `house` is `null` or `house.Walls` is `null`, Gosu throws an NPE exception. Gosu assumes that method calls might have side effects, so Gosu cannot quietly skip the method call and return `null`.

The Gosu support for null-tolerant property accessor paths are a good reason to expose data as *properties* in Gosu classes and interfaces rather than as setter and getter methods.

Primitives and Null-Safe Paths

Gosu null-safety for property paths does not work if the type of the entire expression is a Gosu primitive type. This is equivalent to saying it does not work if the type of the last item in the series of property accesses has a primitive type. For example, suppose you use the property path:

```
a.P1.P2.P3
```

If the type of the `P3` property is `int` or `char` or other primitive types, then Gosu cannot transparently return `null` if any of the following are `null`:

- `a`
- `a.P1`
- `a.P1.P2`

Primitive types (in contrast to object types, which are descendents of `Object`) can never contain the value `null`. Thus Gosu cannot return `null` from that expression, and any casting from `null` to the primitive type would be meaningless. Therefore, Gosu must throw an NPE exception in those conditions.

IMPORTANT Expose public data as properties rather than as getter functions. This allows you to take advantage of Gosu null-safe property accessor paths. Additionally, note it is standard Gosu practice to separate your implementation from your class's interaction with other code by using properties rather than public instance variables. Gosu provides a simple shortcut with the `as` keyword to expose an instance variable as a property. See "Properties" on page 172

Design APIs Around Null Safe Property Paths

You may also want to design your Gosu code logic around this feature. For example, Gosu uses the `java.util.String` class as its native text class. This class includes a built-in method to check whether the `String` is empty. The method is called `isEmpty`, and Gosu exposes this as the `Empty` property. This is difficult to use with Gosu property accessor paths. For example, consider the following `if` statement:

```
if (obj.StringProperty.Empty)
```

Because `null` coerces implicitly to Boolean (the type of the `Empty` property), the expression evaluates to `false` in either of the following cases:

- if `obj.StringProperty` is null
- the `String` is non-null but its `Empty` property evaluates to `false`.

In typical code, it is important to distinguish these two very different conditions cases. For example, if you wanted to use the value `obj.StringProperty` only if the value is non-empty, it is insufficient to just check the value `obj.StringProperty.Empty`.

To work around this, Gosu adds an enhancement property to `java.util.String` called `HasContent`. This effectively is the reverse of the logic of the `Empty` property. The `HasContent` property only returns `true` if it has content. As a result, you can use property accessor paths such as the following:

```
if (obj.StringProperty.HasContent)
```

Because `null` coerces implicitly to Boolean (the type of the `Empty` property), the expression evaluates to `false` in either of the following cases:

- if `obj.StringProperty` is null
- the `String` is non-null but the string has no content (its `HasContent` property evaluates to `false`).

These cases are much more similar semantically than for the variant that uses `Empty` (`obj.StringProperty.Empty`). This means you are more likely to rely on path expressions like this.

Be sure to consider null-safety of property paths as you design your code, particularly with Boolean properties.

IMPORTANT Consider null-safety of property paths as you design your code.

Static Properties

You can use properties directly on the class without creating a new instance of the class. For more information, see "Static Modifier" on page 183.

More Property Examples

The following examples illustrate how to create and use Gosu class properties and get/set methods.

There are two classes, one of which extends the other.

The class myFirstClass:

```
package mypackage

class MyFirstClass {

  // Explicit property getter for Fred
  property get Fred() : String   {
    return "myFirstClass"
  }

}
```

The class mySecondClass:

```
package mypackage

class MySecondClass extends MyFirstClass {

  // Exposes a public F0 property on _f0
  private var _f0 : String as F0

  // Exposes a public read-only F1 property on _f1
  private var _f1 : String as readonly F1

  // Simple variable with explicit property get/set methods
  private var _f2 : String

  // Explicit property getter for _f2
  property get F2() : String {
    return _f2
  }

  // Explicit property setter for _f2, visible only to classes in this package
  internal property set F2( value : String ) {
    _f2 = value
  }

  // A simple calculated property (not a simple accessor)
  property get Calculation() : Number {
    return 88
  }

  // Overrides MyFirstClass's Fred property getter
  property get Fred() : String {
    return super.Fred + " suffix"

}
```

Try the following lines in Gosu Tester to test these classes

First, create an instance of your class:

```
var test = new mypackage.MySecondClass()
```

Assign a property value. This internally calls a hidden method to assign "hello" to variable _f0:

```
test.F0 = "hello"
```

The following line is invalid since f1 is read-only:

```
// This gives a compile error.
test.F1 = "hello"
```

Get a property value. This indirectly calls the mySecondClass property getter function for F2:

```
print( test.F2 ) // prints null because it is not set yet
```

The following line is invalid because F2 is not visible outside of the package namespace of MySecondClass. F2 is publicly read-only.

```
// This gives a compile error.
test.F2 = "hello"
```

Print the Calculation property:

```
print( test.Calculation ) // prints 88
```

The following line is invalid since Calculation is read-only (it does not have a setter function):

```
//This gives a compiler error.
test.Calculation = 123
```

Demonstrate that properties can be overridden through inheritance because properties are virtual:

```
print( test.Fred ) // prints "myFirstClass suffix"
```

Example: Financial Calculations

The following is an example of a sample custom Gosu class that exposes financial calculations as static read-only properties. In this Gosu class:

- A class variable sets the path for `FinancialsCalculationUtil` package so that it does not need to be entered each time.
- The class creates a private static variable for use in constructing customized financial calculations.
- A public method returns the static variable as a read-only value.

Suppose you define this class:

```
package util.financials

class CustomCalcs {

  construct() {
  }

  private static var lib application = gw.api.financials.FinancialsCalculationUtil
  private static var calcMyTotalIncurredNet application = lib.getFinancialsCalculation(
    lib.getGrossTotalIncurredExpression().minus( lib.getTotalRecoveryReservesExpression() ))

  public static property get MyTotalIncurredNet() : FinancialsCalculation {
    return calcMyTotalIncurredNet
  }
}
```

A Guidewire-provided sample rule in the Transaction Approval rule set shows an example usage of the `CustomCalcs` class.

```
var totalIncurredAmt = util.financials.CustomCalcs.MyTotalIncurredNet.getAmount(TransactionSet.Claim)
  if (totalIncurredAmt > 20000) {
    TransactionSet.requireApproval( "Total Incurred on the claim exceeds $20,000" )
}
```

> **Note:** See “Creating Custom Financial Gosu Classes” on page 180 for a discussion on using Gosu classes in financial calculations. See “Using the Pre-defined Financials Calculations” on page 174 for information on the `getAmount()` methods.

Modifiers

There are several types of modifiers:

- Access Modifiers
- Override Modifier
- Abstract Modifier
- Final Modifier
- Static Modifier

Access Modifiers

You can use access modifier keywords to set the level of access to a Gosu class, interface, enumeration, or a type member (a function, variable, or property). The access level determines whether other classes can use a particular variable or invoke a particular function.

For example, methods and variables marked `public` are visible from other classes in the package. Additionally, because they are public, functions and variables also are visible to all subclasses of the class and to all classes outside the current package. For example, the following code uses the `public` access modifier on a class variable:

```
package com.mycompany.utils

class Test1 {
  public var Name : String
}
```

In contrast, the `internal` access modifier lets the variable be accessed only in the same package as the class:

```
package com.mycompany.utils

class Test2 {
  internal var Name : String
}
```

For example, another class with fully qualified name `com.mycompany.utils.Test2` could access the `Name` variable because it is in the same package. Another class `com.mycompany.integration.Test3` cannot see the `Test.Name` variable because it is not in the same package.

Similarly, modifiers can apply to an entire type, such as a Gosu class:

```
package com.mycompany.utils

internal class Test {
  var Name : String
}
```

Some modifiers only apply to type members (functions, variables, properties, and inner types) and some modifiers apply to type members and top-level types (outer Gosu classes, interfaces, enumerations).

The following table lists the Gosu access modifiers and each one's applicability and visibility:

Modifier	Description	Applies to top-level types	Applies to type members	Visible in class	Visible in package	Visible in subclass	Visible by all
`public`	Fully accessible. No restrictions.	Yes	Yes	Yes	Yes	Yes	Yes
`protected`	Accessible only by types with same package and subtypes.	--	Yes	Yes	Yes	Yes	--
`internal`	Accessible only in same package	Yes	Yes	Yes	Yes	--	--
`private`	Accessible only by the declaring type, such as the Gosu class or interface that defines it.	--	Yes	Yes	--	--	--

If you do not specify a modifier, Gosu assumes the following default access levels:

Element	Default modifier
Types / Classes	`public`
Variables	`private`
Functions	`public`
Properties	`public`

Coding Style Recommendations for Variables

Always prefix `private` and `protected` class variables with an underscore character (_).

Also, avoid public variables. If you are tempted to use public variables, convert the public variables to properties. This separates the way other code accesses the properties from the implementation (the storage and retrieval of the properties). For more style guidelines, see "Coding Style", on page 333.

Override Modifier

Apply the `override` modifier to a function or property implementation to declare that the subtype overrides the implementation of an inherited function or property with the same signature.

For example, the following line might appear in a subtype overriding a `myFunction` method in its superclass:

```
override function myFunction(myParameter : String )
```

If Gosu detects that you are overriding an inherited function or method with the same name but you omit the `override` keyword, you get a compiler warning. Additionally, the Gosu editor offers to automatically insert the modifier if it seems appropriate.

Abstract Modifier

The `abstract` modifier indicates that a type is intended only to be a base type of other types. Typically an abstract type does not provide implementations (actual code to perform the function) for some or all of its functions and properties. This modifier applies to classes, interfaces, functions, and properties.

For example, the following is a simple abstract class:

```
abstract class Vehicle {
}
```

If a type is specified as abstract, Gosu code cannot construct an instance of it. For example, you cannot use code such as `new MyType()` with an abstract type. However, you can instantiate a subtype of the type if the subtype fully implements all abstract members (functions and properties). A subtype that contains implementations for all abstract members of its supertype is referred to as a *concrete type*.

For example, if class `A` is abstract and defines one method's parameters and return value but does not provide code for it, that method would be declared `abstract`. Another class `B` could extend `A` and implement that method with real code. The class `A` is the abstract class and the class `B` is a concrete subclass of `A`.

An abstract type may contain implementations for none of its members if desired. This means that you cannot construct an instance of it, although you can define a subtype of it and instantiate that type. For example, suppose you write an abstract Gosu class called `Vehicle` which might contain members but no abstract members, it might look like this:

```
package com.mycompany

abstract class Vehicle {
  var _name : String as Name
}
```

You could not construct an instance of this class, but you could define another class that extends it:

```
package com.mycompany

class Truck extends Vehicle {

  // the subtype can add its own members...
  var _TruckLength : int as TruckLength
}
```

You can now use code such as the following to create an instance of `Truck`:

```
var t = new Truck()
```

Things work differently if the supertype (in this case, `Vehicle`) defines abstract members. If the supertype defines abstract methods or abstract properties, the subtype **must** define an *concrete implementation* of each abstract method or property to instantiate of the subclass. A concrete method implementation must implement actual behavior, not just inherit the method signature. A concrete property implementation must implement actual behavior of getting and setting the property, not just inherit the property's name.

The subtype must implement an abstract function or abstract property with the same name as a supertype. Use the `override` keyword to tell Gosu that the subtype overrides an inherited function or method with the same name. If you omit the `override` keyword, Gosu displays a compiler warning. Additionally, the Gosu editor offers to automatically insert the `override` modifier if it seems appropriate.

For example, suppose you expand the `Vehicle` class with abstract members:

```
package com.mycompany

abstract class Vehicle {

  // an abstract property -- every concrete subtype must implement this!
  abstract property get Plate() : String
  abstract property set Plate(newPlate : String)

  // an abstract function/method -- every concrete subtype must implement this!
  abstract function RegisterWithDMV(registrationURL : String)
}
```

A concrete subtype of this `Vehicle` might look like the following:

```
package com.mycompany

class Truck extends com.mycompany.Vehicle
{
  var _TruckLength : int as TruckLength

  /* create a class instance variable that uses the "as ..." syntax to define a property
   * By doing this, you make a concrete implementation of the abstract property "Plate"
   */
  var _licenseplate : String as Plate

  /* implement the function RegisterWithDMV, which is abstract in your supertype, which
   * means that it doesn't define how to implement the method at all, although it does
   * specify the method signature that you must implement to be allowed to be instantiated with "new"
   */
  override function RegisterWithDMV(registrationURL : String ) {
    // here do whatever needs to be done
    print("Pretending to register " + _licenseplate + " to " + registrationURL)
  }
}
```

You can now construct an instance of the concrete subtype `Truck`, even though you cannot directly construct an instance of the supertype `Vehicle` because it is abstract.

You can test these classes using the following code in the Gosu Tester:

```
var t  = new com.mycompany.Truck()
t.Plate = "GUIDEWIRE"
print("License plate = " + t.Plate)
t.RegisterWithDMV( "http://dmv.ca.gov/register" )
```

This prints the following:

```
License plate = GUIDEWIRE
Pretending to register GUIDEWIRE to http://dmv.ca.gov/register
```

Final Modifier

The `final` modifier applies to types, type members, local variables, and function parameters. It specifies that the value of a property, local variable, or parameter cannot be modified after the **initial** value is assigned. The `final` modifier cannot be combined with the `abstract` modifier on anything. These modifiers are mutually exclusive. The `final` modifier implies that there is a concrete implementation and the `abstract` modifier implies that there is no concrete implementation.

Final Types

If you use the `final` modifier on a type, the type cannot be inherited. For example, if a Gosu class is final, you cannot create any subclass of the final class.

The `final` modifier is implicit with *enumerations*, which are an encapsulated list of enumerated constants, and they are implemented like Gosu classes in most ways. For more information, see “Enumerations”, on page 189. This means that no Gosu code can subclass an enumeration.

Final Functions and Properties

If you use the `final` modifier with a function or a property, the `final` modifier prevents a subtype from overriding that item. For example, a subclass of a Gosu class cannot reimplement a method defined by its superclass if that function is `final`.

For example, suppose you define a class with final functions and properties:

```
package com.mycompany

class Auto {

  // a final property -- no subtype can reimplement / override this!
  final property get Plate() : String
  final property set Plate(newPlate : String)

  // a final function/method -- no concrete subtype can reimplement / override this!
  final function RegisterWithDMV(registrationURL : String)
}
```

In many ways, properties are implemented like functions in that they are defined with code and they are virtual. Being virtual means properties can be overridden and can call an inherited `get` or `set` property function in their supertype. For more information about properties and shortcuts to define properties backed by instance variables, see “Properties” on page 172.

Final Local Variables

You can use the `final` modifier with a local variable to initialize the value and prevent it from changing.

For example, the following code is valid:

```
class final1
{
  function PrintGreeting() {
   var f = "frozen"
f = "dynamic"
  }
}
```

However, this code is not valid:

```
class final1
{
  function PrintGreeting() {
   final var f = "frozen"
   f = "dynamic" // compile error because it attempts to change a final variable
  }
}
```

If you define a variable as final, you must initialize it with a value immediately as you declare the variable. You cannot declare the variable as final and initialize it in a later statement.

Final Function Parameters

You can use the `final` modifier with a function parameter to prevent it from changing within the function.

For example, the following code is valid:

```
package example

class FinalTest
{
  function SuffixTest( greeting : String) {
     greeting = greeting + "fly"
     print(greeting)
  }
}
```

You can test it with the code:

```
var f = new example.FinalTest()
var s = "Butter"
f.SuffixTest( s )
```

This prints:

```
Butterfly
```

However, if you add the `final` modifier to the parameter, the code generates a compile error because the function attempts to modify the value of a final parameter:

```
class final1
{
  function SuffixTest( final greeting : String) {
     greeting = greeting + "fly"
     print(greeting)
  }
}
```

Static Modifier

Static Variables

Gosu classes can define a variable stored once *per Gosu class*, rather than once *per instance* of the class. This can be used with variables and properties. If a class variable is static, it is referred to as a *static variable*.

WARNING If you use static variables in a multi-threaded environment, you must take special precautions to prevent simultaneous access from different threads. Use static variables sparingly if ever. If you use static variables, be sure you understand *synchronized thread access* fully. For more information, see "Concurrency" on page 311.

For additional help on static variables and synchronized access, contact Guidewire Customer Support.

To use a Gosu class variable, remember to set its *access level* such as `internal` or `public` so it is accessible to class that need to use it. For more information access levels, see "Access Modifiers" on page 178.

The `static` modifier cannot be combined with the `abstract` modifier. See "Abstract Modifier" on page 180 for more information.

Static Functions and Properties

The `static` modifier can also be used with functions and properties to indicate that it belongs to the type itself rather than instances of the type.

The following example defines a static property and function:

```
class Greeting {

  private static var _name : String

  static property get Name() : String   {
    return _name
  }

  static property set Name(str : String)   {
    _name = str
  }

  static function PrintGreeting() {
    print("Hello World")
  }
}
```

The `Name` property get and set functions and the `PrintGreeting` method are part of the `Greeting` class itself because they are marked as static.

Consequently, this code in the Gosu Tester accesses properties on the class itself, not an instance of the class:

```
Greeting.Name = "initial value"
print(Greeting.Name)
Greeting.PrintGreeting()
```

Notice that this example never constructs a new instance of the `Greeting` class using the `new` keyword.

Static Inner Types

The `static` modifier can also be used with inner types to indicate that it belongs to the type itself (the class itself) rather than a specific instance of the type.

The following example defines a static *inner class* called `FrenchGreeting` within the `Greeting` class:

```
package example

class Greeting
{
  static class FrenchGreeting {
     static public function sayWhat() : String {
       return "Bonjour"
     }
  }

  static public property get Hello() : String {
    return FrenchGreeting.sayWhat()
  }
}
```

You can test this in the Gosu Tester using the code:

```
print(example.Greeting.Hello)
```

This prints:

```
Boujour
```

For more information about this topic, refer to the next section, “Inner Classes” on page 184.

Inner Classes

You can define inner classes in Gosu, similar to inner classes in Java. They are useful to for encapsulating code even further and defining it in the same file as related code. Use named inner classes if you want to be able to refer to the inner class from multiple related methods or multiple related classes. Use anonymous inner classes if you just need a simple subclass that you can define in-line within a class method.

Inner classes optionally can include generics features (see “Gosu Generics”, on page 221).

Named Inner Classes

You can define a named class within another Gosu class. Once defined, it can be used within the class within which it is defined, or from classes that derive from it. If using it from the current class,

The following example defines a static *inner class* called `FrenchGreeting` within the `Greeting` class:

```
package example

class Greeting
{
  static class FrenchGreeting {
     static public function sayWhat() : String {
       return "bonjour"
     }
  }

  static public property get Hello() : String {
    return FrenchGreeting.sayWhat()
  }
}
```

You can test this in the Gosu Tester using the code:

```
print(example.Greeting.Hello)
```

This prints:

```
bonjour
```

Notice that this example never constructs a new instance of the `Greeting` class or the `FrenchGreeting` class using the `new` keyword. The inner class in this example has the `static` modifier. For more information the static modifier, see “Static Modifier” on page 183.

Similarly, classes that derive from the outer class can use the inner class `FrenchGreeting`. The following example subclasses the `Greeting` class:

```
package example

class OtherGreeting extends Greeting
{
  public function  greetme () {
    var f = new Greeting.FrenchGreeting()
    print(f.sayWhat())
  }
}
```

You can test this code using the following code in the Gosu Tester:

```
var t = new example.OtherGreeting()
t.greetme()
```

This prints:

```
bonjour
```

Anonymous Inner Classes

You can define anonymous inner classes in Gosu from within a class method, similar to usage in Java. The syntax for creating an anonymous inner class is very different from a named inner class. Anonymous inner classes are similar in many ways to creating instances of a class with the `new` operator. However, you can extend a base class by following the class name with braces and then add additional variables or methods. If you do not have another useful base class, use `Object`.

The following is a class that uses an anonymous inner class:

```
package example

class InnerTest {

  static public function runme() {

    // create instance of an anonymous inner class that derives from Object
    var counter = new Object() {

      // anonymous inner classes can have variables (public, private, and so on)
      private var i = 0

      // anonymous inner classes can have constructors
      construct() {
        print("Value is " + i + " at creation!")
      }

      // anonymous inner classes can have methods
      public function incrementMe () {
        i = i + 1
        print("Value is " + i)
      }
    }

    // "counter" is a variable containing an instance of a
    // class that has no name, but derives from Object and
    // adds a private variable and a method

    counter.incrementMe()
    counter.incrementMe()
    counter.incrementMe()
    counter.incrementMe()
    counter.incrementMe()
  }
}
```

You can use the following code in the Gosu Tester to test this class:

```
example.InnerTest.runme()
```

This prints:

```
Value is 0 at creation!
Value is 1
Value is 2
Value is 3
Value is 4
Value is 5
```

Example: Advanced Anonymous Inner Class

The following example shows how to use an anonymous inner class that derives from a more interesting object than `Object`. In this example, the constructor and another method are inherited by the new inner class.

Suppose you define a base class for your inner class and call it `Vehicle`:

```
package example

class Vehicle
{
  construct()
  {
     print("A vehicle was just constructed!")
  }

  function actionOne(s : String) {
     print("actionOne was called with arg " + s)
  }
}
```

You can create a different class that uses `Vehicle` and defines an anonymous inner class based on `Vehicle`:

```
package example

class FancyVehicle
{

  public function testInner() {

    // Create an inner anonymous class that extends Vehicle
    var test = new Vehicle() {
        public function actionTwo(s : String) {
          print("actionTwo was called with arg " + s)
        }
      }
      test.actionOne( "USA" )
      test.actionTwo( "GUIDEWIRE" )
    }
}
```

Notice that the inner class that defines the `actionTwo` method uses the `new` operator and not the `class` operator. What it actually does, however, is define a new class with no name and then creates one instance of it.

You can test the `FancyVehicle` class with the following code in Gosu Tester:

```
var g = new example.FancyVehicle()
g.testInner()
```

This prints:

```
A vehicle was just constructed!
actionOne was called with arg USA
actionTwo was called with arg GUIDEWIRE
```

Gosu Block Shortcut for Anonymous Inner Classes Implementing an Interface

If the anonymous inner class implements an interface and the interface has **exactly one method**, then you can use a Gosu block to implement the interface as a block. This is an alternative to using an explicit anonymous class. This is true for interfaces originally implemented in either Gosu or Java.

The parameters of the block are the same number and type as the parameters to the single method. The return type is the same as the return type of that method. This feature works with any interface, including interfaces defined as inner interfaces within a class.

This Gosu block shortcut is helpful for writing concise code in some situations. For example, APIs that use the Java interface called `Runnable`, which is a simple container for code within a method called `run`.

For example, suppose the `PluginCallbackHandler` class contains an inner interface called `PluginCallbackHandler.Block`, which implements a `run` method, similar to the `Runnable` interface. This interface has one method. Instead of creating an anonymous class to use the inner interface, use a block that takes no arguments and has no return value.

For example, suppose you are using this `PluginCallbackHandler` class definition in Java:

```
public interface PluginCallbackHandler {

  // DEFINE AN INNER INTERFACE WITHIN THIS CLASS
  public interface Block {
  public void run() throws Throwable;
  }

// ...

public void execute(Block block) throws Throwable;
}
```

This Gosu code creates the anonymous class explicitly:

```
public function messageReceived(final messageId : int) : void {

   // CREATE AN ANONYMOUS CLASS THAT IMPLEMENTS THE INTERFACE
   var myBlock : PluginCallbackHandler.Block = new PluginCallbackHandler.Block() {

      // implement the run() method in the interface
      public function run() : void { /* your Gosu statements here */ }
    };

    // pass the anonymous inner class with the one method
    _callbackHandler.execute(myBlock);
  }
```

However, you can write it more concisely using Gosu block syntax:

```
  public function messageReceived(messageId : int) {

    _callbackHandler.execute(\ -> { /* your Gosu statements here */ })
}
```

For more information about blocks, see “Gosu Blocks” on page 213.

chapter 10

Enumerations

An enumeration is a list of named constants that are encapsulated into a special type of class. Gosu supports enumerations natively, as well as provides compatibility to use enumerations defined in Java.

This topic includes:

Using Enumerations

An enumeration is a list of named constants that are encapsulated into a special type of class. For example, an application tracking cars might want to store the car manufacturer in a property, but track them as named constants that can be checked at compile-time. Gosu supports enumerations natively and also is compatible with enumerations defined in Java.

To create an enumeration

1. Create a class by that name using the same approach you use to create a class.

   For example, in Studio, right-click on a package folder and choose the **New** submenu, followed by the **Class** menu item to create the class in that package (in that namespace). Studio creates an initial version of the class such as:

   ```
   package example

   class FruitType {
     construct() {
     }
   }
   ```

2. Change the keyword `class` to `enum` and remove the constructor. Your enumeration now looks like:

   ```
   package example

   enum FruitType {

   }
   ```

3. Add your named constants separated by commas:

```
enum FruitType {
  Apple, Orange, Banana, Kiwi, Passionfruit
}
```

Extracting Information from Enumerations

To use the enumerations, simply reference elements of the enumeration class:

```
uses example.FruitType
var myFruitType = FruitType.Banana
```

To extract the name of the enumeration value as a `String`, get its `Name` property. To extract the index of the enumeration value as an `Integer`, get its `Ordinal` property.

For example:

```
print(myFruitType.Name) // prints "Banana"

print(myFruitType.Code) // prints "Banana"

print(myFruitType.Ordinal) // prints "2"
```

Comparing Enumerations

You can compare two enumerations using the == operator. For example,

```
if (myFruitType == FruitType.Apple)
  print("An apple a day keeps the doctor away.")

if (myFruitType == FruitType.Banana)
  print("Watch out for banana peels.")
```

chapter 11

Interfaces

Gosu can define and implement *interfaces* that define a strict contract of interaction and expectation between two or more software elements. From a syntax perspective, interfaces look like class definitions but merely specify a set of required functions necessary for any class that implements the interface. An interface is conceptually a list of method signatures grouped together. Some other piece of code must implement that set of methods to successfully implement that interface. Gosu classes can implement interfaces defined in either Gosu or Java.

This topic includes:

What is an Interface?

Interfaces are a set of required functions necessary for a specific task. Interfaces define a strict contract of interaction and expectation between two or more software elements, while leaving the implementation details to the code that implements the interface. In many cases, the person who writes the interface is different from the person who writes code to implement the interface.

To take a real-world example of an interface, imagine a car stereo system. The buttons, such as for channel up and channel down, are the interface between you and the complex electrical circuits on the inside of the box. You press buttons to change the channel. However, you probably do not care about the implementation details of how the stereo performs those tasks behind the solid walls of the stereo. If you get a new stereo, it has equivalent buttons and matching behavior. Since you interact only with the buttons and the output audio, if the user interface is appropriate and outputs appropriate sounds, the internal details do not matter to you. You do not care about the details of how the stereo internally handles the button presses for channel up, channel down, and volume up.

Similarly, ClaimCenter defines interfaces that ClaimCenter calls to perform various tasks or calculate values. For example, to integrate ClaimCenter with a document management system, implement a plugin interface that defines how the application interacts with a document management system. The **implementation details** of document management are separate from the contract that defines what actions your document management code must handle. For more information about plugins, see "Plugin Overview", on page 101.

If a Gosu class implements this interface, Gosu validates at compile time that all required methods are present and that the implementor class has the required method signatures.

An interface appears like a group of related method signatures with empty bodies grouped together for the purpose of some other piece of code implementing the methods. if a class implements the interface, the class agrees to implement all these methods with the appropriate method signatures. The code implementing the interface agrees that each method appropriately performs the desired task if external code calls those methods.

You can write Gosu classes that implement or extend interfaces defined in Gosu or defined in Java.

Defining and Using an Interface

In some ways, interfaces are similar to Gosu classes.

To create an interface in Guidewire Studio, first create an appropriate package. There is no separate top-level folder in Studio for interfaces. Right-click on a package folder, then click **New** → **Interface**. Studio creates a new interface with the appropriate syntax for you.

Then, write the rest of the interface like a Gosu class, except that methods are method signatures only with no method bodies. For example, define a simple interface with the following code:

```
interface Restaurant {
  function retrieveMeals() : String[]
  function retrieveMealDetails(dishname : String) : String
}
```

To implement an interface, create a different Gosu class and add "`implements` *`MyInterfaceName`*" after the class name. For example, if your class is called `MyRestaurant`, go to the line:

```
class MyRestaurant
```

Change that line to:

```
class MyRestaurant implements Restaurant
```

If a class implements more than one interface, separate the interface names by commas:

```
class MyRestaurant implements Restaurant, InitializablePlugin
```

In the example `Restaurant` interface, you can implement the interface with a class such as:

```
class MyRestaurant implements Restaurant

  override function retrieveMeals() {
    return {"chicken", "beef", "fish"}
}
  override function retrieveMealDetails(mainitem : String) : String {
    return "Steaming hot " + dishname + " on rice, with a side of asparagus."
  }
}
```

The Gosu editor reveals compilation errors if your class does not properly implement the plugin interface. You must fix these issues.

A common compilation issue is that a method that interface methods that look like properties must be implemented in Gosu explicitly as a Gosu property. In other words, if the interface contains a method whose name starts with `"get"` or `"is"` and takes no parameters, define the method using the Gosu `property` syntax. In this case, do not use the `function` keyword to define it as a standard class method.

For example, if interface `IMyInterface` declares methods `isVisible()` and `getName()`, your plugin implementation of this interface might look like:

```
class MyClass implements IMyInterface {

  property get Visible() : Boolean {
  ...
  }
  property get Name() : String {
  ...
  }
}
```

For more information about properties, see “Defining and Using Properties with Interfaces” on page 193.

If desired, you can write Gosu interfaces that extend from Java interfaces. You can also have your interface include Gosu generics. Your class can extend from Java classes that support generics. Your class can abstract an interface across a type defined in Java or a subtype of such a type. (For more information about generics, see “Gosu Generics”, on page 221.)

Defining and Using Properties with Interfaces

Interfaces created in Gosu can declare properties. This means that you can define explicit `property get` or `property set` accessors in interfaces with the following syntax:

```
property get Description() : String
```

Classes can implement an interface property with the explicit `property get` or `property set` syntax.

For example, if the interface is defined as:

```
package example

interface MyInterface
{
  property get VolumeLevel() : int
  property set VolumeLevel(vol : int) : void
}
```

A class could implement this interface with this code:

```
class MyStereo implements MyInterface
{
   var _volume : int

   property set VolumeLevel(vol : int) {
     _volume = vol
   }

   property get VolumeLevel() : int {
    return _volume
  }
}
```

You can test this code in the Gosu tester:

```
uses example.MyStereo

var v = new MyStereo()
v.VolumeLevel = 11
print("the volume goes to " + v.VolumeLevel)
```

If you run this code, it prints:

```
the volume goes to 11
```

Alternatively, a class implementing a property can implement the property using the *variable alias* syntax using the `as` keyword. This language feature lets you make simple get and set methods that use an class instance variable to store the value, and to get the value if anyone requests it.

For example, the following code is functionally identical to the previous example implementation of `MyStereo`, but it is much more concise:

```
uses example.MyStereo
class MyStereo implements MyInterface
{
   var _volume : int as VolumeLevel
}
```

If you run the Gosu tester code as before, it prints the same results.

For information about Gosu class properties in general, see “Classes”, on page 169.

Interface Methods that Look Like Properties

If an interface’s methods look like properties, a class implementing an interface must implement the interface in Gosu as a Gosu property using with `property get` or `property set` syntax. In other words, if the interface contains a method whose name starts with "`get`" or "`is`" and takes no parameters, define the method using the Gosu `property` syntax. See earlier in this section for examples.

Modifiers and Interfaces

In many ways, interfaces are defined like classes. One way in which they are similar is the support for modifier keywords. For more information on modifiers, see “Modifiers” on page 178.

One notable differences for interfaces is that the `abstract` modifier is implicit for the interface itself and all methods defined on the interface. Consequently, you cannot use the `final` modifier on the interface or its members.

Superclass Properties

When implementing an interface and referencing a superclasses’ property, use the `super.`*`PropertyName`* syntax, such as:

```
property get Bar() : String {
... _mySpecialPrivateVar = super.Foo + super.Bar
  }
```

chapter 12

Composition

Gosu supports the language feature called *composition* using the `delegate` keyword in variable definitions. Composition allows a class to delegate responsibility for implementing an interface to a different object. This compositional model allows easy implementation of objects that are proxies for other objects, or encapsulating shared code independent of the type inheritance hierarchy.

This topic makes extensive references to the following topics:

- "Interfaces" on page 191
- "Classes" on page 169

This topic includes:

Using Gosu Composition

The language feature *composition* allows a class to delegate responsibility for implementing an interface to a different object. This feature helps reuse code easily for some types of projects with complex requirements for shared code. With composition, you do not rely on class inheritance hierarchies to choose where to implement reusable shared code.

Class inheritance is useful for some types of programming problems. However, it can make complex code dependencies fragile. Class inheritance tightly couples a base class and all subclasses. This means that changes to a base class can easily break all subclasses classes. Languages that support multiple inheritance (allowing a type to extend from multiple supertypes) can increase such fragility. For this reason, Gosu does not support multiple inheritance.

What if you have shared behavior that applies to multiple unrelated classes? Since they are unrelated, class inheritance does not naturally apply. Classes with a shared behavior or capability might **not** share a common type inheritance ancestor other than `Object`. Because of this, there is no natural place to implement code that applies to both classes.

Let us consider a general example to illustrate this situation. Suppose you have a window class and a clipboard-support class. Suppose you have a user interface system with different types of objects and capabilities. However, some of the capabilities might not correspond directly to the class inheritance. For example, suppose you have classes for visual items like windows and buttons and scroll bars. However, only some of these items might interact with the clipboard copy and paste commands.

If not all user interface items do not support the clipboard, you might not want to implement your clipboard-supporting code in the root class for your user interface items. However, where do you put the clipboard-related code if you want to write a window-handling class that is also a clipboard part? One way to do this is to define a new interface that describes what methods each class must implement to support clipboard behavior. Each class that uses this interface implements the interface with behavior uniquely appropriate to each class. This is an example of sharing a behavioral contract defined by the interface. However, each implementation is different within each class implementation.

What if the actual implementation code for the clipboard part is identical for each class that uses this shared behavior? Ideally, you write shared code only **once** so you have maximum encapsulation and minimal duplication of code. In some cases there does not exist a shared root class other than `Object`, so it might not be an option to put the code there. If Gosu supported multiple inheritance, you could encapsulate the shared code in its own class and classes could inherit from that class in addition to any other supertype.

Fortunately, you can get many of the benefits of multiple inheritance using another design pattern called *composition*. Composition encapsulates implementation code for shared behavior such that calling a method on the main object forwards method invocations to a subobject to handle the methods required by the interface.

Let us use our previous example with clipboard parts and windows. Let us suppose you want to create a subclass of window but that implements the behaviors associated with a clipboard part. First, create an interface that describes the required methods that you expect a clipboard-supporting object to support, and call it `IClipboardPart`. Next, create an implementation class that implements that interface, and call it `ClipboardPart`. Next, create a window subclass that implements the interface and delegates the actual work to a `ClipboardPart` instance associated with your window subclass.

The delegation step requires the Gosu keyword `delegate` within your class variable definitions. Declaring a delegate is like declaring a special type of class variable.

The `delegate` keyword has the following syntax:

```
delegate PRIVATE_VARIABLE_NAME represents INTERFACE_LIST
```

Or optionally

```
delegate PRIVATE_VARIABLE_NAME : TYPE represents INTERFACE_LIST
```

The *`INTERFACE_LIST`* is a list of one or more interface names, with commas separating multiple interfaces.

For example:

```
delegate _clipboardPart represents IClipboardPart
```

Within the class constructor, create an instance of an object that implements the interface. For example:

```
construct() {
  _clipboardPart = new ClipboardPart( this )
}
```

After that point in time, Gosu intercepts any method invocations on the object for that interface and forward the method invocation to the delegated object.

Let us look at complete code for this example.

The interface:

```
package test

interface IClipboardPart
{
  function canCopy() : boolean
  function copy() : void
  function canPaste() : boolean
  function paste() : void
```

```
}
```

The delegate implementation class:

```
package test

class ClipboardPart implements IClipboardPart {
  var _myOwner : Object

  construct(owner : Object) {
    _myOwner = owner
  }

  // this is an ACTUAL implementation of these methods...
  override function canCopy() : boolean { return true }
  override function copy() : void { print("Copied!")}
  override function canPaste() : boolean { return true }
  override function paste() : void { print("Pasted!") }

}
```

Your class that delegates the `IClipboardPart` implementation to another class

```
package test

class MyWindow implements IClipboardPart {
  delegate _clipboardPart represents IClipboardPart

  construct() {
    _clipboardPart = new ClipboardPart( this )
  }
}
```

Finally, enter the following code into the Gosu Tester:

```
uses test.MyWindow

var a = new MyWindow()

// call a method handled on the delegate
a.paste()
```

It prints:

```
Pasted!
```

Overriding Methods Independent of the Delegate Class

You can override any of the interface methods that you delegated. Using the previous example, if the `canCopy` method is in the delegate interface, your `MyWindow` class can choose to override the `canCopy` method to specially handle it. For example, you could trigger different code or choose whether to delegate that method call.

For example, your `MyWindow` class can override a method implementation using the override keyword, and calling the private variable for your delegate if desired:

```
override function canCopy() : boolean
{
  return someCondition && _clipboardPart.canCopy();
}
```

Declaring Delegate Implementation Type in the Variable Definition

You can declare a delegate with an explicit type for the implementation class. This is particularly valuable if any of your code accessing the delegate directly in terms of the implementation class. For example, by declaring the type explicitly, you can avoid casting before calling methods on the implementation class that you know are not defined in the interface it implements.

To declare the type directly, add the implementation type name followed by the keyword `represents` before the interface name. In other words, use the following syntax:

```
private delegate PRIVATE_VARIABLE_NAME : IMPLEMENTATION_CLASS represents INTERFACE_NAME
```

For example,

```
private delegate _clipboardPart : ClipboardPart represents IClipboardPart
```

Using One Delegate for Multiple Interfaces

You can use a delegate to represent (handle methods for) multiple interfaces for the enclosing class. Instead of providing a single interface name, specify a comma-separated list of interfaces. For example:

```
private delegate _employee represents ISalariedEmployee, IOfficer
```

You might notice that in this example the line does not specify an explicit type for `_employee` and yet it represents **two** different types (in this case, two interface types). You might wonder about the compile-time type of the variable called `_employee`. Because the variable must satisfy all requirements of both types, Gosu uses a special type called a *compound type*. A literal of this type is expressed in Gosu as a list separated by the ampersand symbol (&). For example:

```
ISalariedEmployee & IOfficer
```

Typical code does not need to mention a compound type explicitly. However, remember this syntax in case you see it during debugging code that uses the `delegate` keyword with multiple interfaces.

For more details of compound types, see "Compound Types" on page 308.

Using Composition With Built-in Interfaces

You can use composition with any interfaces, including built-in interfaces. For example, you could give a custom object all the methods of `java.util.List` and delegate the implementation to an instance of `java.util.ArrayList` or another `List` implementation.

For example:

```
class MyStringList implements List<String>
{
  delegate _internalList represents List<String> = new ArrayList<String>()
}
```

You could now use this class and call any method defined on the `List` interface:

```
var x = new MyStringList()
x.add( "TestString" )
```

chapter 13

Annotations and Interceptors

Gosu annotations are a simple syntax to provide metadata about a Gosu class, constructor, method or property. This annotation can control the behavior of the class, the documentation for the class. For more advanced usage, annotations can define an *interceptor* for a method. This allows you to wrap a method with code before the method runs, after the method runs, or both. This enables certain types of *aspect programming*.

This topic includes:

Annotating a Class, Method, Type, or Constructor

Annotations are a simple syntax to add metadata to a Gosu class, constructor, method, or property. For example, annotations could add indicate what a method returns, or indicate what kinds of exceptions the method might throw. You can add completely custom annotations and this information can be read at run time. If you use an annotation, use the at sign (`@`), followed by the annotation name, immediately before declarations of what they annotate.

For example, the following simple example specifies a class to expose as a web service for external systems:

```
@WebService
class MyServiceAPI {
  public function myRemoteMethod() {}
}
```

In some cases, you follow the annotation name with an argument list within parentheses. The following example specifies a function might throw a specific exception using arguments to the annotation:

```
class MyClass{

@Throws(java.text.ParseException, "If text is invalid format, throws ParseException")
  public function myMethod() {}
}
```

The annotation may not require any arguments, or the arguments may be optional. If so, you can omit the parentheses. For example, suppose you add an annotation called `MyAnnotation` that takes no arguments. You could use it in the following (verbose) syntax:

```
@MyAnnotation()
```

Since there are no arguments, you can optionally omit the parentheses:

```
@MyAnnotation
```

You can use annotations defined natively in Gosu or directly use Java annotations.

Argument List Notes

Gosu requires argument lists to be in the same format as regular function or method argument lists:

```
// standard Gosu argument lists
@KnownBreak("user", "branch", "ABC-xxxxx")
```

Gosu does not support the special named arguments calling convention from Java:

```
// invalid in Gosu
@KnownBreak(targetUser = "user", targetBranch = "branch", jira = "ABC-xxxxx")
```

Built-in Annotations

The Gosu language includes built-in annotations defined in the `gw.lang.*` package, which is always in scope, so their fully-qualified name is not required.

The following table lists the built-in general annotations:

Annotation	Description	Usage limits	Parameters
`@Param`	Specifies the documentation of a parameter.	Methods only	(1) The name of the parameter. (2) Documentation in Javadoc format for the method's parameter.
`@Returns`	Specifies the documentation for the return result of the method.	Methods only, but only once per method	(1) Documentation in Javadoc format for the method's return value.
`@Throws`	Specifies what exceptions might be thrown by this method.	Methods only	(1) An exception type. (2) A description in Javadoc format of what circumstances it would throw that exception, and how to interpret that exception.
`@Deprecated`	Specifies not to use a class, method, constructor, or property. It goes away in a future release. Begin rewriting code to avoid using this class, method, constructor, or property.	Can appear anywhere, but only once for any specific class, method, constructor, or property.	(1) A warning string to display if this deprecated class, method, or constructor is used.

The following code defines a class that uses several built-in annotations:

```
package com.mycompany
uses java.lang.Exception

@WebService
class Test
{
  @Param("Name", "The user's name. Must not be an empty string.")
  @Returns("A friendly greeting with the user's name")
  @Throws(Exception, "General exception if the string passed to us is empty or null")
  public function FriendlyGreeting(Name : String) : String {

    if (Name == null or Name.length == 0) throw "Requires a non-empty string!"

    return "Hello, " + Name + "!"
  }
}
```

The following example specifies that a method is *deprecated*, which means it was a valid API but not anymore. A deprecated API is temporarily possibly to use but it is absent in a future release. Immediately start rewriting code that uses a deprecated API.

```
class MyClass {

  @Deprecated("Don't use MyClass.myMethod(). Instead, use betterMethod()")
  public function myMethod() {print("Hello")}

  public function betterMethod() {print("Hello, World!")}
}
```

Because annotations are implemented as Gosu classes (see "Defining Your Own Annotations" on page 202), the annotation class that you are implicitly using must be in the current Gosu scope. You can ensure that it is in scope by fully qualifying the annotation. For example, if the `SomeAnnotation` annotation is defined within the package `com.mycompany.some.package`, specify the annotation like:

```
@com.mycompany.some.package.SomeAnnotation
class SomeClass {
  ...
}
```

Alternatively, import the package using the Gosu `uses` statement and then use the annotation more naturally and concisely by using only its name:

```
uses com.mycompany.some.package.SomeAnnotation.*

@SomeAnnotation
class SomeClass {
  ...
}
```

There are also several built-in annotations related to publishing Gosu classes as web services, which are discussed further in the "Publishing a Web Service" on page 32:

Annotation	Description	Usage limits	Parameters
`@WebService`	Specifies a class to export to external systems as a web service (a SOAP API). For details, see "Web Services (SOAP)" on page 25 in the *Integration Guide*.	Classes only	There are various optional parameters. See "Publishing a Web Service" on page 32 for details.
`@WebServiceMethod`	Enables specific authentication rules and restrictions to be enforced on a per-method level instead of at a class level. Whenever there is an authentication set for both the method and class, the method annotation takes precedence. For details, see "Publishing a Web Service" on page 32.	Methods only	There are required parameters. See "Publishing a Web Service" on page 32 for details.
`@DoNotPublish`	A method-level annotation for Gosu classes. Use this on public methods of SOAP web services not to publish in the web service.	Methods only	No parameters. See "Publishing a Web Service" on page 32 for details.

There are also annotations that are reserved for internal use. You may see the following in built-in Gosu code. These are **unsupported** for you to write. The following table lists internal annotations only so that you understand their role in Studio modules other than `configuration`.

Internal annotation	Description	Usage limits
`@Export`	Let a class be visible and editable in Studio	Classes only
`@ReadOnly`	Let a class be visible in Studio but non-editable. This means it does not permit copying it into the `configuration` module for you to modify.	Classes only

Annotations at Run Time

You can get annotation information from a class either directly by getting the type from an object at runtime. You can get an object's type at runtime using the `typeof` operator, such as: `typeof` *`TYPE`*

You can get annotation information from a type, a constructor, a method, or a property by accessing their type information objects attached to the type. You can call the `getAnnotation` method to get all instances of specific annotation, as a list of annotation instances. In the examples in the table, the variable `i` represents the index in the list. In practice, you would probably search for it by name using `List` methods like `list.firstWhere(\ s -> s.Name = "MethodName")`.

Get annotations on a specific instance of a...	Example using the @Deprecated annotation
Type	`(typeof obj).TypeInfo.getAnnotation(Deprecated)`
Constructor	`(typeof obj).TypeInfo.Constructors[i].getAnnotation(Deprecated)`
Method	`(typeof obj).TypeInfo.Methods[i].getAnnotation(Deprecated)`
Property	`(typeof obj).TypeInfo.Properties[i].getAnnotation(Deprecated)`

Using these methods, the return result is automatically statically typed as a list of the proper type. Using the examples in the previous table, the result would be of type:

```
List<Deprecated>
```

This type is shown using generics syntax, and it means "a list of instances of the `Deprecated` annotation class". For more information about generics, see "Gosu Generics", on page 221.

You can additionally get all annotations (not just one annotation type) using the two properties `Annotations` and `DeclaredAnnotations`. These two properties are slightly different and resemble the Java versions of annotations with the same name. On types and interfaces, `Annotations` returns all annotations on this type/interface and on all its supertypes/superinterfaces. `DeclaredAnnotations` returns annotations only on the given types, ignoring supertypes/superinterfaces. In constructors, properties, and methods, the `Annotations` and `DeclaredAnnotations` properties return the same thing: all annotations including supertypes/superinterfaces. In the examples in the table, the variable `i` represents the index in the list. In practice, you would probably search for it by name using `List` methods like `list.firstWhere(\ s -> s.Name = "MethodName")`.

Get all annotations on...	Example
Type	`(typeof obj).TypeInfo.Annotations`
Constructor	`(typeof obj).TypeInfo.Constructors[i].Annotations`
Method	`(typeof obj).TypeInfo.Methods[i].Annotations`
Property	`(typeof obj).TypeInfo.Properties[i].Annotations`

For a detailed example of accessing annotations at run time, see "Defining Your Own Annotations" on page 202.

Defining Your Own Annotations

You can define new annotations to add entirely new metadata annotations, apply them to various kinds of programming declarations, and then retrieve this information at run time. You can also get information at run time about objects annotated with built-in annotations. For example, you could mark a Gosu class with metadata and retrieve it at run time.

Annotations are implemented as Gosu classes, and an annotation is simply a call to the annotation class's constructor. A class constructor is similar to a class method. However, Gosu automatically calls the constructor if it creates a new instance of the class, such as if Gosu code uses the `new` keyword.

You can define new annotation types that can be used throughout Gosu. Annotations are defined just like classes except they must extend the interface `IAnnotation`. The `IAnnotation` interface is a marker interface that designates a class as an *annotation definition*.

Suppose you want a new annotation that allows us to annotate which people wrote a Gosu class. You could use the annotation at run time for debugging information or to file a bug in certain error conditions. To do this, you can create an annotation called `Author`.

For example, the following example defines a new annotation `Author` in the `com.mycompany` package

```
package com.mycompany

class Author implements IAnnotation {
}
```

In this case the annotation has no constructor, which implies the annotation takes no parameters, so it could simply be called as:

```
@Author()
```

Because there are no arguments, you can omit the parentheses:

```
@Author
```

However, as written in this example so far, you used the annotation but not specified any authors. Annotations can define arguments so you can pass information to the annotation, which might stored in private variables. Annotations can have properties or arguments of any type. However, if defining properties and arguments, be careful you never define circular references between annotation classes and regular classes.

This example requires only a single `String` argument, so define the annotation `Author` to take one argument to its constructor. Gosu calls the constructor once for the type after initializing Gosu at run time. In your constructor, save the constructor arguments value in a private variable:

```
package com.guidewire.pl.docexamples.annotations

class Author implements IAnnotation
{
  // Define a public property Author, backed by private var named _author
  private var _author : String as AuthorName

  construct(a : String)
  {
    // The constructor takes a String, which means the Author of this item
    _author = a;
  }
}
```

In this example, the annotation saves the `String` argument in a class instance variable called `_author`. Because of the phrase “`as Author`” in the definition of the variable, at run time you can extract this information as the annotation’s public property `Author`.

By default, this annotation can be used on any method, type, property, or constructor, and as many times as desired. For example, you could specify multiple authors for a class or even multiple authors for methods on a class, or both. You can customize these settings and restrict the annotation’s usage, as discussed in “Customizing Annotation Usage” on page 204.

For now, test this annotation by using it on a newly defined type, such as a new Gosu class. Create the following class in the `com.mycompany` package:

```
package com.guidewire.pl.docexamples.annotations

uses com.guidewire.pl.docexamples.annotations.Author

@Author("A. C. Clarke")
@Author("J. M. Straczynski")
class Spaceship {

}
```

You can get annotation information from a class either directly by getting the type from an object at runtime. First can get an object’s type at runtime using the `typeof` operator or by getting the `Type` property from an object. Next, get its `TypeInfo` property and call the `getAnnotation` method, passing your annotation class name directly as a parameter. The result is a list (`java.util.List`) containing annotation information objects. For each one of these, get its `Value` property and coerce it to your annotation class.

For example, add the example classes from earlier in this topic into your Gosu environment and then paste the following code into the Gosu Tester:

```
uses com.guidewire.pl.docexamples.annotations.Author
uses com.guidewire.pl.docexamples.annotations.Spaceship

var sub = new Spaceship()

print("Get annotations from an object's type, then iterate with 'for'...")
var annotations = (typeof sub).TypeInfo.getAnnotation(Author)
for (a in annotations)  {
  print("  Author: " + (a.Value as Author).AuthorName);
}

print("")
print("Get annotations directly from a type, then iterate with a block...")
var annotations2 = Spaceship.Type.TypeInfo.getAnnotation(Author)
annotations2.each( \ a -> print("  Author: " + (a.Value as Author).AuthorName))
```

This example prints the following:

```
Get annotations from an object's type, then iterate with 'for'...
  Author: A. C. Clarke
  Author: J. M. Straczynski

Get annotations directly from a type, then iterate with blocks...
  Author: A. C. Clarke
  Author: J. M. Straczynski
```

For more information about blocks and using collections, see "Gosu Blocks", on page 213 and "Collections", on page 231.

Customizing Annotation Usage

Usage of each annotation can be customized, such as allowing it under certain conditions. For example, notice that the built-in annotation `@Returns` can appear only on methods. To restrict, usage like this, use the `AnnotationUsage` meta-annotation within your annotation definition. The `AnnotationUsage` meta-annotation takes two parameters, the *target* and the *modifier*.

The target defines where the annotation can be used using these enumerations:

- `annotation.UsageTarget.MethodTarget` - This annotation can be used on a method.
- `annotation.UsageTarget.TypeTarget` - This annotation can be used on a type, including classes
- `annotation.UsageTarget.PropertyTarget` - This annotation can be used on a property.
- `annotation.UsageTarget.ConstructorTarget` - This annotation can be used on a constructor.
- `annotation.UsageTarget.AllTarget` - This annotation can be used on all of the above contexts.

The modifier defines how many times the annotation can be used (for that target) using these enumerations:

- `annotation.UsageModifier.None` - This annotation cannot exist on that target
- `annotation.UsageModifier.One` - This annotation can only appear once on that target
- `annotation.UsageModifier.Many` - This annotation can appear many (unlimited) times on that target

For example, the `@Returns` annotation can only appear on methods, and can only appear once, so it specifies its requirements with this line right before its annotation definition:

```
@AnnotationUsage(annotation.UsageTarget.Method, annotation.UsageModifier.One)
```

The default availability is universal. In other words, if no `AnnotationUsage` attribute is defined on an annotation, the usage defaults to allow the annotation **unlimited** times on **all** parts of a type or class.

However, once any `AnnotationUsage` annotation is used in an annotation definition, all targets default to `None`. After using `AnnotationUsage` once, Gosu requires you to explicitly specify supported targets using `AnnotationUsage` meta-annotations. You can optionally add multiple lines for each type of permitted use.

> **IMPORTANT** The default annotation availability is universal (all parts, many times). As soon as you use one `AnnotationUsage` line in the annotation definition, Gosu assumes all targets revert to `None.` Explicitly list all permitted usages with `AnnotationUsage` lines as appropriate.

The `annotation` class is always in scope. You do not need to fully-qualify the class name or use a `uses` statement in files that use it.

Gosu Interceptors

Gosu also supports a programming language feature called *interceptors* that allows you to wrap calls to a class's methods and constructors with additional logic defined in Gosu. The feature is sometimes called a *cross cut aspect*.

To use this feature, define a new annotation class that specifies a new annotation. This new annotation indicates class methods that Gosu intercepts if any code calls them. Once intercepted, Gosu executes code that you define in the interceptor and optionally passes arguments included on the annotation definition. In addition to implementing the `IAnnotation` interface, the new annotation class must implement one or both of the interfaces `IPreMethodInterceptor` and `IPostMethodInterceptor`.

You can control what aspect of the method call chain to override:

- intercept before calling the method (the *pre-method* interceptor)
- intercept after calling the method (the *post-method* interceptor)
- intercept both before and after calling the method (both pre-method and post-method interceptors)

If you specified `IPreMethodInterceptor`, implement the following method within your interceptor class:

```
public function onBeforeMethod(instance : Object, methodName : String, args : Object[]) { ... }
```

The `onBeforeMethod` method defines special code to execute **before** the method is called, and it takes the following arguments:

- the instance (the Gosu class instance acted upon)
- the method name (as a `String`)
- a list of method arguments for the intercepted method.

> **Note:** If the method to intercept is a class constructor (the special function called immediately after an object is created), Gosu calls the pre-method interceptor *before* calling the constructor. In this case, the instance value for `onBeforeMethod` is always `null`.

If you specified `IPostMethodInterceptor`, implement the following method within your interceptor class:

```
public function onAfterMethod(instance : Object, methodName : String, args : Object[]) { ... }
```

The `onAfterMethod` method defines special code to execute **after** the method is called, and it takes the following arguments:

- the instance (the Gosu class instance acted upon)
- the method name (as a `String`)
- a list of other method arguments for the intercepted method

For example, suppose you want to intercept a certain method call and add code both before and after the method call. The following code defines a wrapper called `Logme` that intercepts methods both before and after the method call if you apply the annotation to a method. This code prints to standard out a basic debugging notification message:

```
package com.mycompany

// This annotation takes one parameter, a String, which is printed in the log message
class Logme implements IAnnotation, IPreMethodInterceptor, IPostMethodInterceptor {

  public function onBeforeMethod(instance : Object, methodName : String, args : Object[]) {
    print("")
    print( DateTime.CurrentDate.toString())
    print(" METHOD CALL " + (typeof instance).Name + "." + methodName )
    print("  Arguments: ")
    args.each( \ arg -> print("   " + arg) )
  }

  public function onAfterMethod(instance : Object, methodName : String, args : Object[],
           returnValue : Object) {
    print(DateTime.CurrentDate.toString())
    print(" RETURNED FROM  " + (typeof instance).Name + "." + methodName  )
    print("  Return value was: " + returnValue)

  }
}
```

To use this wrapper, apply the annotation before methods in a new class. For example:

```
package com.mycompany

class Submarine
{
  @Logme
   function verticalChange(distance : int) {
     if (distance != 0) {
        print("vertical change " + distance + " ... going " + (distance > 0 ? "up" : "down"))
     }
     for (var i in 5000000) {
       // do nothing for a while; note the timestamps printed out by Logme before and after!s
     }
}

  @Logme
   function periscopeAction( movePeriscopeUp : Boolean) : String {
     print("periscope " + (movePeriscopeUp ? "up!" : "down!"))

     if (movePeriscopeUp) {
       return "View is good!"
     }
     else {
       return "Dark!"
     }
   }
}
```

You can test this code by pasting the following in the Gosu Tester:

```
uses com.mycompany.Submarine

var s = new Submarine()
s.verticalChange( -5 )
s.periscopeAction( true )
```

If any code calls the function `verticalChange`, then the output is:

```
Mon Apr 07 21:03:29 PDT 2008
 METHOD CALL com.mycompany.Submarine.verticalChange( int )
  Arguments:
   -5
vertical change = -5   (going down)
Mon Apr 07 21:03:31 PDT 2008
 RETURNED FROM  com.mycompany.Submarine.verticalChange( int )
  Return value was: null

Mon Apr 07 21:03:31 PDT 2008
 METHOD CALL com.mycompany.Submarine.periscopeAction( java.lang.Boolean )
  Arguments:
   true
periscope up!
Mon Apr 07 21:03:31 PDT 2008
 RETURNED FROM  com.mycompany.Submarine.periscopeAction( java.lang.Boolean )
  Return value was: View is good!
```

The example output contains much information about the intercepted method calls, including:

- the types of objects
- the method name
- the types of parameters the method takes
- the actual parameters passed to the method
- the return values of the method captured and printed *after* the method completes
- timestamps

Using information such as this, you can create your own types of contextual or debugging information. However, be careful of performance issues. Anticipate potential performance issues in your interceptor design. Test the effects thoroughly once you implement the interceptors. Ensure that your code performs well with your final interceptor code.

WARNING If you use interceptors, beware of potential performance implications. Test your interceptor performance carefully.

chapter 14

Enhancements

Gosu enhancements are a language feature that allows you to augment classes and other types with additional concrete methods and properties. For example, use enhancements to define additional utility methods on a class or interface that cannot be directly modified, even code written in Java. You can enhance Guidewire entity instances and classes originally written in Gosu or Java. *Enhancing* is different from *subclassing* in important ways. Enhancing a class makes new methods and properties available to **all** objects of that enhanced type, not just Gosu code that explicitly knows about the subclass. Use enhancements to add powerful functionality omitted by the original authors.

This topic includes:

Using Enhancements

Gosu *enhancements* allow you to augment classes and other types with additional concrete methods and properties. The most valuable use of this feature is to define additional utility methods on a Java class or interface that cannot be directly modified. This is most useful if a class's source code is unavailable, or a given class is *final* (cannot be subclassed). Enhancements can be used with interfaces as well as classes, which means you can add useful methods to interfaces.

Enhancing a class or other type is different from subclassing: enhancing a class makes the new methods and properties available to **all** instances of that class, not merely subclass instances. For example, if you add an enhancement method to the `String` class, all `String` instances in Gosu automatically have the additional method.

You can also use enhancements to overcome the language shortcomings of Java or other languages defining a class or interface. For example, Java-based classes and interfaces can be used from Gosu, but they do not natively allow use of blocks, which are anonymous functions defined in-line within another function. (See "Gosu Blocks", on page 213.) Guidewire includes many built-in enhancements to commonly-used Java classes in its products so that any Gosu code can use them.

For example, Gosu extends the Java class `java.util.ArrayList` so you can use concise Gosu syntax to sort, find, and map members of a list. These list enhancements add additional methods to Java lists that take Gosu blocks as parameters. The original Java class does not support blocks because the Java language does not support blocks. However, these enhancements add utilities without direct modifications to the class. Gosu makes these additional methods automatically and universally available for all places where Gosu code uses `java.util.ArrayList`.

You can also enhance an interface. This does **not** mean an enhancement can add new methods to the interface itself. The enhancement does not add new requirements for classes to implement the interface. Instead, enhancing an interface means that all objects whose class implements the interface now has new methods and properties. For example, if you enhance the `java.util.Collection` interface with a new method, all collection types suddenly have your newly-added method.

Although you can add custom properties to existing entities, any new properties that you add do not appear in the generated Data Dictionary documentation that describes the application data model.

This does not go into detail about the built-in enhancements to collections. For reference documentation, see “Collections”, on page 231. If you have not yet learned about Gosu *blocks*, you may want to first review “Gosu Blocks”, on page 213.

Syntax for Using Enhancements

There is no special syntax for using an already-defined enhancement. The new methods and properties are automatically available within the Gosu editor for all Gosu contexts.

For example, suppose there is an enhancement on the `String` type for an additional method called `calculateHash`. Use typical method syntax to call the method with any `String` object accessible from Gosu:

```
var s1 = "a string"
var r1 = s1.calculateHash()
```

You could even use the method on a value you provide at compile time:

```
"a string".calculateHash()
```

Similarly, if the enhancement adds a property called `MyProperty` to the `String` type, you could use code such as:

```
var p = "a string".MyProperty
```

The new methods and properties all appear in the list of methods that appears if you type a period (.) character in the Gosu editor. For example, if typing “`s1.calculateHash()`”, after you type “`s1.`” the list that appears displays the `calculateHash` method as a choice.

Creating a New Enhancement

To create a new enhancement, put the file in your Gosu class file hierarchy in the package that represents the enhancement. It does not need to match the package of the enhanced type.

In Studio, right-click on a package folder, then click **New → Enhancement**. A dialog appears and you can enter the enhancement class name and the type (typically a class name or entity type name) that you are enhancing. Studio creates a new enhancement with the appropriate syntax for you.

Syntax for Defining Enhancements

Although using enhanced properties and methods is straightforward, a special syntax is necessary for defining new enhancements. Defining a new Gosu enhancement looks similar to defining a new Gosu class, with some minor differences in their basic definition.

Differences between classes and enhancements include

- Use the keyword `enhancement` instead of `class`
- To define what to enhance, use the syntax: “`: TYPETOEXTEND`” instead of “`extends CLASSTOEXTEND`”

- If you must reference methods on the enhanced class/type, use the symbol `this` to see the enhanced class/type. For example, to call the enhanced object's `myAction` method, use the syntax `this.myAction()`. In contrast, never use the keyword `super` in an enhancement.

  Note: Enhancements technically are defined in terms of the *external interface* of the enhanced type. The keyword `super` implies a superclass rather than an interface, so it is inappropriate for enhancements.

- Enhancements cannot save state information by allocating new variables or properties on the enhanced type.

Enhancement methods can use properties already defined on the enhanced object or call other enhanced methods.

You can add new *properties* as necessary and access the properties on the class/type within Gosu. However, that does not actually allow you to save state information for the enhancement unless you can do so using variables or properties that already exist on the enhanced type. See later in this section for more on this topic.

Note: Although you can add custom properties to Guidewire entity types, any new properties you add do not appear in the generated Data Dictionary documentation that describes the application data model.

For example, the following enhancement adds one standard method to the basic `String` class and one property:

```
package example

enhancement StringTestEnhancement : java.lang.String {

  public function myMethod(): String {
    return "Secret message!"
  }

  public property get myProperty() : String {
    return "length : " + this.length()
  }
}
```

Note the use of the syntax "`property get`" for the method defined as a property.

With this example, use code like the following to get values:

```
// get an enhancement property:
print("This is my string".myProperty)

// get an enhancement method:
print("This is my string".myMethod())
```

These lines outputs the following:

```
"length: 17"
"Secret message!"
```

Enhanced methods can call other methods internally, as demonstrated with the `getPrettyLengthString` method, which calls the built-in `String` method `length()`.

IMPORTANT Enhancements can create new methods but cannot override existing methods.

Setting Properties in Enhancements

Within enhancement methods, your code can set other values as appropriate such as an existing class instance variable. You can also set properties with the "`property set` *`PROPERTYNAME`*`()`" syntax. For example, this enhancement creates a new settable property that appends an item to a list:

```
package example

enhancement ListTestEnhancement<T> : java.util.ArrayList<T>
{
  public property set LastItem(item : T) {
    this.add(item)
  }
}
```

Test this code in the Gosu Tester with this code:

```
uses java.util.ArrayList
```

```
var strlist = new ArrayList<String>() {"abc", "def", "ghi", "jkl"}

print(strlist)
strlist.LastItem = "hello"
print(strlist)
```

This code outputs:

```
[abc, def, ghi, jkl]
[abc, def, ghi, jkl, hello]
```

You can add new properties and add property set functions to set those properties. However, in contrast to a class, enhancements cannot define **new** variables on the type to store instance data for your enhancement. This limits most types of state management if you cannot directly change the source code for the enhanced type to add more variables to the enhanced type. Enhancements cannot add new variables because different types have dramatically different property storage techniques, such as a persistent database storage, Gosu memory storage, or file-based storage. Enhancements cannot transparently mirror these storage mechanisms.

Also, although enhancements can add properties, enhancements cannot override existing properties.

> **IMPORTANT** Enhancements can add new properties by adding new dynamic property get and set functions to the type. However, enhancements cannot override property get or set functions. Also, enhancements cannot create new native variables on the object that would require additional data storage with the original object. Enhancements cannot override methods either.

Enhancement Naming and Package Conventions

The name of your enhancement must follow the following naming convention of the enhanced type name, then an optional functional description, followed by word `Enhancement`. In other words, the format is:

```
[EnhancedTypeName][OptionalFunctionalDescripton]Enhancement
```

For example, to enhance the `Report` class, you could call it simply:

```
ReportEnhancement
```

If the enhancement added methods related to claim financials, you might emphasize the enhancement's functional purpose by naming the enhancement:

```
ReportFinancialsEnhancement
```

Enhancement Packages

Use your own company package to hierarchically group your own code and separate it from built-in types, in almost all cases. For example, you could define your enhancement with the fully-qualified name `com.mycompany.ReportEnhancement`. Even if you are enhancing a built-in type, if at all possible use your own package for the enhancement class itself.

In only extremely rare cases, you might need to enhance a built-in type and you need to use a `protected` property or method. If so, you might need to define your enhancement in a subpackage of the enhanced type. See “Modifiers” on page 178 for more information about the `protected` keyword. However, to avoid namespace conflicts with Guidewire types, avoid this approach if possible.

Enhancements on Arrays

To specify the enhanced type for an enhancement on an array type:

- For regular types, use standard array syntax, such as `String[]`.
- For generic types, use the syntax `T[]`, which effectively means all arrays.

chapter 15

Gosu Blocks

Gosu blocks are a special type of function that you can define in-line within another function. You can then pass that block of code to yet other functions to invoke as appropriate. Blocks are very useful for generalizing algorithms and simplifying interfaces to certain APIs. For example, blocks can simplify tasks related to collections, such as finding items within or iterating across all items in a collection.

This topic includes:

What Are Blocks?

Gosu blocks are functions without names (sometimes called *anonymous functions*) that you can define in-line within another function. You can then pass that block of code to yet other functions to invoke as appropriate. Blocks can be very useful for generalizing algorithms and simplifying interfaces to APIs. An API author can design most of an algorithm but let the API consumer contribute short blocks of code to complete the task. The API can use this block of code and call it once or possibly many times with different arguments.

For example, you might want to find items within a collection that meet some criteria, or to sort a collection of objects by certain properties. If you can describe your find or sort criteria using small amount of Gosu code, Gosu takes care of the general algorithm such as sorting the collection.

Some other programming languages have similar features and call them *closures* or *lambda expressions*. For those who use the Java language, notice that Gosu blocks serve some most common uses of single-method anonymous classes in Java. However, Gosu blocks provide a concise and clear syntax that makes this feature more convenient in typical cases.

Blocks are particularly valuable for the following:

- **Collection manipulation.** Using collection functions such as `map` and `each` with Gosu blocks allows concise easy-to-understand code with powerful and useful behaviors for real-world programming.
- **Callbacks.** For APIs that wish to use callback functions after an action is complete, blocks provide a straightforward mechanism for triggering the callback code.
- **Resource control.** Blocks can be useful for encapsulating code related to connection management or transaction management. (See “Bundles and Transactions”, on page 275.)

Gosu code using blocks appropriately can simplify and reduce the size of your Gosu code. However, they can also be confusing if used too aggressively and use them carefully. If your intended use does not fall into one of the list categories, reconsider whether to use blocks. There may be a better and more conventional way to solve the problem. Generally speaking, if you write a method that takes more than one block as a function/method argument, strongly consider redesigning or refactoring the code.

WARNING Gosu blocks are not always the correct design solution. For example, if you design a function that takes more than one block as arguments, a general rule is to redesign or refactor your code.

Basic Block Definition and Invocation

To define a Gosu block, type use the backslash character (\) followed by a series of arguments. The arguments must be name/type pairs separated by a colon character (`:`) just as if defining arguments in a method. Next, add a hyphen character (-) and a greater-than character (>) to form the arrow-like pair characters ->. Finally, add a Gosu expression or a statement list surrounded by curly braces.

In other words, the syntax is:

```
\ argumentList -> blockBody
```

The argument list (*`argumentList`*) is a standard function argument list, for example:

```
x : Number, y : Number
```

The argument list defines what parameters must be passed to the block. The parameter list uses identical syntax as parameters to regular functions. However, in some cases you can omit the types of the parameters, such as passing a block directly into a class method such that the parameter type can be inferred. For examples, see “Argument Type Inference Shortcut In Certain Cases” on page 217.

The block body (*`blockBody`*) can be either of the following:

- a simple expression. This includes anything legal on the **right-hand** side of an assignment statement. For example, the following is a simple expression:
  ```
  "a concatenated string " + "is a simple expression"
  ```
- a statement list with one or more statements surrounded by braces and separated by semi-colon characters, such as the following simple one-statement statement list:
  ```
  \ x : Number, y : Number -> { return x + y }
  ```
  For single-statement statement lists, you *must* explicitly include the brace characters. In particular, note that variable assignment operations are always statements not expressions. Thus, the following expression is invalid:
  ```
  names.each( \ n -> myValue += n )
  ```
  Instead, change it to the following:
  ```
  names.each( \ n -> { myValue += n } )
  ```
  For multiple statements, separate the statements with a semi-colon character. For example:
  ```
  \ x : Number, y : Number -> { var i = x + y; return i }
  ```

The following block multiplies a number with itself, which is known as squaring a number:

```
var square = \ x : Number-> x * x     //no need for braces here
var myResult = square(10)          // call the block
```

The value of `myResult` in this example is 100.

> **IMPORTANT** All parameter names in a block definition's argument list must not conflict with any existing in-scope variables, including but not limited to local variables.

The Gosu editor displays a block definition's backslash character as a Greek lambda character. This improves code appearance and honors the theoretical framework from which blocks derive, called *lambda calculus*. The Gosu editor displays the pair of characters -> as an arrow symbol.

For example, you could type the following Gosu block:

```
var square = \ x : Number -> x * x
```

The Gosu editor displays it as:

```
var square = λ x : Number → x
```

Guidewire strongly recommends omitting semicolon characters in Gosu at the end of lines because code is more readable without optional semicolons. However, if providing statement lists on one line, such as within block definitions, use semicolon characters to separate statements. (For other style guidelines, see "General Coding Guidelines" on page 333.)

Return Values and Return Type

Notice that the block definition does not explicitly declare the *return type*, which is the type of the return value of the block. This is because the return type is inferred from either the expression (if you defined the block with an expression) or for statement list by examining the `return` statements. This frees you of the burden of explicitly typing the return type. This also allows the block to appear short and elegant. However, it is important to understand that the return type is actually *statically typed* even though the type is not explicitly visible in the code.

For example, note the following simple block:

```
var blockWithStatementBody = \ -> { return "hello blocks" }
```

Because the statement `return "hello blocks"` returns a `String`, that means the block's return type is `String`.

> **IMPORTANT** Gosu infers a block's return type by the returned value of the return statements of the statement list. If an expression is provided instead of a statement list, Gosu uses the type of the expression. That type is static (fixed) at compile time although it is not explicitly visible in the code.

Using and Invoking Blocks

Blocks are invoked just like normal functions by referencing a variable to which you previously assigned the block. To use a block, type:

1. the name of the block variable or an expression that resolves to a block
2. an open parenthesis
3. a series of argument expressions
4. a closing parenthesis

For example, suppose you create a Gosu block with no arguments and a simple return statement:

```
var blockWithStatementBody = \-> { return "hello blocks" }
```

Because the statement list returns a `String`, Gosu infers that the block returns a `String`. The new block is assigned to a new variable `blockWithStatementBody`, and the block has a return type of `String` even though this fact is not explicit in the code text.

To call this block and assign the result to variable `myresult`, simply use this code:

```
var myresult = blockWithStatementBody()
```

The value of the variable `myresult` is the `String` value `"hello blocks"` after this line executes:

The following example creates a simple block that adds two numbers as parameters and returns the result:

```
var adder = \ x : Number, y : Number -> { return x + y }
```

After defining this block, you can call it with code such as:

```
var mysum = adder(10, 20)
```

The variable `mysum` has the type `Number` and has the value 30 after the line is executed.

You can also implement the same block behavior by using an expression rather than a statement list, which allows an even more concise syntax:

```
var adder = \ x : Number, y : Number -> x + y
```

Variable Scope and Capturing Variables In Blocks

Gosu blocks maintain some context with respect to the enclosing statement in which they were created. If code in the block refers to variables that are defined outside the scope of the block's definition but in scope where the block is defined, the variable is *captured*. The variable is incorporated **by reference** into the block. Incorporating the variable by reference means that blocks do not merely capture the current value of the variable at the time its enclosing code creates the block. If the variable changes after the enclosing code creates the block, the block gets or sets the most recent value in the *original* scope. This is true even if the *original* scope exited (finished).

The following example adds 10 to a value. However, the value 10 was captured in a local variable, rather than included in an argument. The captured variable (called `captured` in this example) is used but not defined within the block:

```
var captured = 10
var addTo10 = \ x : Number -> captured + x
var myresult = addTo10(10)
```

After the third line is executed, `myresult` contains the value 20.

A block captures the state of the stack at the point of its declaration, including all variables and the special symbol `this`, which represents the current object. For example, the current instance of a Gosu class running a method.

This capturing feature allows the block to access variables in scope *at its definition*:

- ...even after being passed as an argument to another function
- ...even after the block returns to the function that defines it
- ...even if some code assigns it to a variable and keeps it around indefinitely
- ...even after the original scope exits (after it finishes)

In other words, each time the block runs, it can access all variables declared in that original scope in which it was defined. The block can get or set those variables. The values of captured variables are evaluated each time the block is executed, and can be read or set as desired. Captured variable values are **not** simply a static snapshot of their value at the time the block was created.

To illustrate this point further, the following example creates a block that captures a variable (`x`) from the surrounding scope. Next, the code that created the block changes the value of `x`. Only after that change does any code actually call the block:

```
// define a local variable, which is captured by a block
var x = 10

// create the block
var captureX = \ y : Number -> x + y

// Note: the variable "x" is now SHARED by the block and the surrounding context
```

```
// Now change the value of "x"
x = 20

// at the time the block runs, it uses the current value of x,
// this is NOT a snapshot of what it was at the time block was created
var z = captureX( 10 )

print(z) // prints 30   --- not 20!!!
```

The captured variable is effectively **shared** by the original scope and the block that was created within that scope. In other words, the block references the variable itself, not merely its original value.

IMPORTANT If accessing variables not defined within the block definition, blocks effectively share the variable with the context that created it. This is true even if the original scope exited (finished) or its value has changed. This is a very powerful feature. If you use this feature at all, use it very carefully and document your assumptions so people who read your code can understand and debug it.

Argument Type Inference Shortcut In Certain Cases

The Gosu parser provides additional type inference in a common case. If a block is defined within a method call parameter list, Gosu can infer the type of the block's arguments from the parameter argument. You do not need to explicitly specify the argument type in this case.

In other words, if you pass a block to a method, in some cases Gosu can infer the type so you can omit it for more concise code. This is particularly relevant for using collection-related code that takes blocks as arguments.

For example, suppose you had this code:

```
var x = new ArrayList<String>(){"a", "b", "c"}

var y = x.map( \ str : String -> str.length )
```

You could instead omit the argument type (`String`). The `map` method signature allows Gosu to infer the argument type in the block because of how the `map` method is defined.

You could use the more concise code:

```
var x = new ArrayList<String>(){"a", "b", "c"}

var y = x.map( \ str -> str.length )
```

The list method `map()` is a built-in list enhancement method that takes a block with one argument. That argument is always the same as the list's type. Therefore Gosu infers that `str` must be of type `String` and the you do not need to explicitly define the type of arguments nor the return type.

Note: The `map` method is implemented using a built-in Gosu enhancement of the Java language `List` class. For more information, see "Collections", on page 231.

Block Type Literals

Block literals are a form of type literal, which means the way you reference a *block type*. The block literal specifically what kinds of arguments the block takes and what type of return value it returns.

Block Types In Declarations

If you define a variable to contain a block in a variable declaration, the preferred syntax is:

```
variableName( list_of_types ) : return_type
```

For example, to declare that `x` is a variable that can contain a block that takes a single `String` argument and returns a `String` value, use this code:

```
var x( String ) : String
```

In declarations, you can also optionally use the `block` keyword, although this is discouraged in declarations:

```
block( list_of_types ) : return_type
```

For example, this code declares the same block type as described earlier:

```
var x : block( String ) : String
```

Block Types Not Part of Declarations

Where a block type literal is **not** part of a declaration, the `block` keyword is strictly required:

```
block( list_of_types ) : return_type
```

For example:

```
var b = block( String ) : Number
```

This means that the `b` variable is **assigned** a value that is a block type. Since the block type literal is not directly part of the declaration, the `block` keyword must be specified.

Block Types In Argument Lists

Within function definition, a function argument can be a block. As you define the block argument, provide a **name** for that block parameter so you can use it within the function. Do this using the following syntax for block types in argument lists:

```
parameter_variable_name( list_of_types ) : return_type
```

For example, suppose you want to declare a function that took one argument, which is a block. Suppose the block takes a single `String` argument and returns no value. If you want refer to this block by name as `myCallback`, define the argument using the syntax:

```
myCallBack( String ) : void
```

It might be easier to understand with an actual example. The following Gosu class includes a function that takes a callback block. The argument is called *`myCallBack`*, which is a block that takes a single string argument and returns no value. The outer function calls that callback function with a `String`.

```
package mytest

class test1 {
  function myMethod( myCallBack( String ) : void ) {

    // call your callback block and pass it a String argument
    myCallBack("Hello World")

  }
}
```

Test this code as follows:

```
var a = new mytest.test1()
a.myMethod( \ s : String -> print("<contents>" + s + "</contents>") )
```

For even more concise code, you can omit the argument type "`: String`" in the in-line block. The block is defined in-line as an argument to a method whose argument types are already defined. In other words, you can simply use the following code

```
var a = new mytest.test1()
a.myMethod( \ s -> print("<contents>" + s + "</contents>") )
```

Both versions print the following:

```
<contents>Hello World</contents>
```

Block Types BNF Notation

For those interested in formal BNF notation, the notation of a block literal is:

```
blockliteral -> block_literal_1 | block_literal_2
block_literal_1 -> block ( type_list ) : type
block_literal_2 -> parameter_name ( type_list ) : return_type
type_list -> type | type_list , | null
```

Blocks and Collections

Gosu blocks are particularly valuable for working with collections of objects. Blocks allow concise and easy-to-understand code that loops across items, extracts information from every item in a collection, or sorts items. Common collection enhancement methods that use blocks are `map`, `each`, and `sortby`.

For example, suppose you want to sort the following list of strings:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}
```

You could easily resort the list based on the length of the strings using blocks. Create a block that takes a `String` and returns the sort key, which in this case is the string’s length. The built-in list `sortBy(...)` method handles the rest of the sorting algorithm and then returns the new sorted array:

```
var resortedStrings = myStrings.sortBy( \ str -> str.Length() )
```

These block-based collection methods are implemented using a built-in Gosu enhancement of the Java language `List` class. For more information, see “Collections”, on page 231.

Blocks as Shortcuts for Anonymous Classes

You can use a block in place of an anonymous class that implements an interface with a single method. For more information, see “Classes” on page 169.

chapter 16

Gosu Generics

Gosu generics is a language feature that lets you define a class or function as working with many types by abstracting its behavior across multiple types of objects. This abstraction feature is important because collections defined with generics can specify what kinds of objects they contain. If you use collections, you can be specific about what objects are in the collection. You do not need to very general about the type of the contents, such as using a root type such as `Object`. However, if designing APIs that can work with different types of objects, you can write the code only once and it works with different types of collections. In essence, you can generalize class or methods to work with various types and retain compile-time type safety. Use generics to write statically typed code that can be abstracted to work with multiple types.

Generics are especially valuable for defining special relationships between arguments to a function and/or its return values. For example, you can require two arguments to a function to be homogenous collections of the same type of object and the function returns the same type of collection. Designing APIs to be abstract like that allows your code and the Gosu language to infer other relationships. For example, an API that returns the first item in a collection of `String` objects is always typed as a `String`. You need not write coercion code with the syntax "`as` *`TYPE`*" as often if designing your APIs to use generics. Because generics increase how often Gosu can use *type inference*, your collection-related code can be easy-to-understand, concise, and type-safe.

Gosu generics are compatible with generics in Java version 1.5, so you can use Java classes designed for Java 1.5 generics or even extend them in Gosu.

For more information about static typing in Gosu, see "More About the Gosu Type System" on page 27.

This topic includes:

Gosu Generics Overview

You probably use simple arrays sometimes to store multiple objects of the same type. For example, an array of five numbers, an array of forty-seven `String` objects, or an array of some other type of primitive or object. Similarly, *collections* (including all *lists*) provide another way of grouping items together but with important differences between arrays and collections.

Standard arrays contain items of the same type of object and if one type extends another, you can make certain assumptions about the type of items in the array. For example, because `Integer` extends `Number`, it means that an array of `Integer` is also an array of `Number`. In other words, `Integer[]` is also an array of `Number[]`. Where a `Number[]` is required**,** you are free to pass or assign an `Integer[]`. However, collections do not work that way.

Standard collections can contain a variety of types of objects (they are *heterogeneous*). If you take an object out of a collection, typically you must cast it to a desired type or check its type before doing something useful with it.

Without generics, in practice, people tend to design collection-related APIs to work with collections of `Object` instances. The `Object` class is the root class of all non-primitive objects in the Gosu language and also in the Java language.

Unfortunately, if you use APIs that return collections of type `Object`, your code must cast (coerce) it to a more specific type to access properties and methods.

Although casting one value to another type is useful, it is unsafe in some cases and prevents the compiler from knowing at compile-time whether it succeeds. For example, if the item you extract from the collection is a type that does not support casting, it fails at **run time**. For example, casting a `String` to an `Array`. This approach to coding is inconsistent with confirming type problems at **compile time**. Detecting problems at compile time is important for reliable mission-critical server applications.

A simple alternative is to define different types of collections/lists that support only homogenous sets of objects of a certain type, such as a `StringList`, an `IntegerList`, or a `MyCustomClassList`. This provides compile-time certainty and thus dramatically reduces the chance of type safety issues. However, the downside is more complexity to make the API work with different types of lists. A Gosu class method that takes a `StringList` would need a separate method signature to take an `IntegerList`. This type of repetitive method signature declaration simply to achieve type safety is time consuming. Additionally, it might be incomplete: if you provide an API, it cannot predict list of types you do not know about that a consumer of your API wants to use. If only there were a way to generalize the function so that it would work with all lists, you could provide a generalized (or generic) function to perform the task.

Suppose you could define a collection with an explicit type of each item. With the angle-bracket notation after the collection class such as `List<Number>`, you can specify what types of things the container contains. If you read aloud, you can translate the bracket notation in English as the word "of". Thus, in English, the syntax `List<Number>` means "a list of numbers". Even better, suppose there was a way to define function parameters to work with **any type**. What if it always returns an object of the same type, or an array of that type, and have such relationships enforced at compile time? This is what Gosu generics do for you.

Generics provide a way to specify the **type of objects in a collection** with specificity as you use an API but with generality as you design an API. At compile time, the Gosu compiler confirm everything has a valid type for the API. Additionally, Gosu **infers types** of arguments and return values in many cases so you do not have to do much coercing values from a root class. For instance, you do not generally need to coerce a variable of type `Object` to a more useful specific type. Suppose you take values out of a collection of objects of type `MyClass`. A variable that contains an extracted first item in the collection always has type `MyClass`, not `Object`. With generics you do not need to coerce the value to type `MyClass` before calling methods specific to the `MyClass` class.

Generics provide the best of *generalizability* as you design APIs and *specificity* as you use APIs. Using generics, your collections-related code can be easy-to-understand, concise, and typesafe.

Gosu generics are compatible with generics implemented in Java version 1.5. You can use Java utility classes designed for Java 1.5 generics and even extended them in Gosu. There is one exception for Java-Gosu compatibility, which is that Gosu does not support the syntax `<? super TYPE>`. For more information about other similar features, see "Bounded Wildcards" on page 226.

For extended discussions of generics as implemented in Java, see the book "Java Generics and Collections" by Maurice Naftalin and Philip Wadler, or the following on-line tutorial:

```
http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
```

Gosu Generics are Reified

One important difference between Gosu and Java is that Gosu generics are *reified*. This means that unlike Java, at run time, Gosu retains the actual specific type. In other words, at run time you could check whether an object was an instance of `PetGroup<Cat>` or `PetGroup<Dog>` including the information in the angle brackets.

In contrast, Java generics lose this generic parameter information at run time. This is called *type erasure*. Java introduced generics in this way to maximize compatibility with older Java code that did not support generics.

Using Gosu Generics

If a function or method has already defined arguments or return value using Gosu generics, as a "consumer" of this API, the API is easy to use. The only important thing to know is that you define the type of collection with the angle bracket notation `COLLECTION_CLASS<OF_TYPE>`. For example, an array list of `Address` objects would use the syntax `ArrayList<Address>`.

> **Note:** In practice, you sometimes do not need to define the collection type due to type inference or special object constructors. See "Basic Lists" on page 231 and "Basic HashMaps" on page 233.

For example, suppose you want a list of `String` objects. One way to define the list would be:

```
var theList= new ArrayList<String>() { address1, address2, address3, address4 }
```

You could create a function that takes a specific type of list, in this case a list of strings:

```
function printStrings( strs : ArrayList<String> ) {
  for( s in strs ) {
  print( s )
  }
}
```

If you want to call a method using Gosu generics to take an array list of any type, simply call the method:

```
r = printStrings(theList)
```

If Gosu knows the return result type of a function, it can infer the type of other things, which makes your code more concise:

```
var strs = new ArrayList<String>(){"a", "ab", "abc"}
var longerStrings = strs.findAll( \ str -> str.length >= 2 )
```

In the previous example, the resulting value of `longerStrings` is strongly typed as `ArrayList<String>`. This is because Gosu knows the return type of the `findAll` method called on any array list of `String` values. If you get an object from value of `longerStrings`, it has the correct expected `String` type, not simply `Object`.

Using functions defined with generics typically are simple, often even more simple because of Gosu generics. The return value can be **strongly typed** to match the exact type of collection you passed into it or one of its items. For example, return a "list of `MyClass` objects" or "a `MyClass` object", rather than a "list of `Object`" or just an `Object`. Although generics are abstract, using APIs other people defined with generics typically is straightforward and intuitive.

The Power of Generics Comes From Wildcards

Although you can specify a specific type of collection, the greatest power of Gosu generics is defining APIs that work with multiple types, not just a single type. This requires a special syntax using wildcards.

Without Gosu generics, the way to support multiple types would be define a utility class method that takes a standard `Collection` object and returns another `Collection` object. That would allow you to use the method with a wide variety of collections: a Collection of MyClass objects, a Collection of Address objects, and so on. However, any code that extracted items from the collection after an API call would have to add code with a coercion "x as TYPE" if extracting an object from it:

```
var f = myCollection.iterator.next()
(f as MyClass).myClassMethod()
```

Note the code "f as MyClass". That approach typically results in hard-to-read code, since you must manually add casting to a specific type for a variety of APIs due to this issue. Additionally, it is dangerous because the actual casting happens at run time and you could make a mistake by casting it to the wrong object. Most importantly, the casting could **fail at run time** in some cases if you make other types of errors, rather than identified and flagged at compile time. Fortunately, with generics this type of casting is not necessary if you use APIs designed with generics and design any new APIs with generics.

That would allow us to remove the cast (the "as MyClass") from the previous example:

```
var a = new ArrayList<MyClass>() { c1, c2, c3 }
...

// the result of this is strongly typed
var first = myResults.iterator.next().myClassMethod
```

From quickly looking at the code, you might assume from the text that the first variable is not strongly typed after removing the cast. However, it is strongly typed at compile time.

If you want to make full use of the language's ability to use generic types, you have two choices:

- *parameterize* a class, which means to add generic types to the class definition
- *parameterize* a method, which means to add generic types to a method definition

Parameterized Classes

If you want a class that always operates with a generic type, define the class with the angle bracket notation CLASSNAME<*GENERIC_TYPE_NAME*> in the class definition. By convention, for the *GENERIC_TYPE_NAME* string, use a one-letter variable, preferably T. For example, you could define a class MyClass as MyClass<T>.

In the following example, the class Genericstest has one method that returns the first item in a list. Gosu strongly types the return value to match the type of the items in the collection:

```
package com.example
uses java.util.ArrayList

class Genericstest<T>
{
  // print out (for debugging) and then return the first item in the list, strongly typed
  public function PrintAndReturnFirst(aList : ArrayList<T>) : T {
    print(aList[0])
    return aList[0]
  }
}
```

Now, some other code could use this class and pass it an array list of any type:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}

var t = new Genericstest<String>()
var first = t.PrintAndReturnFirst( myStrings )
```

After this code runs, the value of the variable first is **strongly typed** as String because of how it used the method that was defined with generics.

This also works with multiple dimensions of types. Suppose you want to write something that stores key-value maps. Instead of writing:

```
class Mymapping {
  function put( key : Object, value : Object) {...}
  function get( key : Object) : Object {...}
}
```

...you could use generics to define it as:

```
class Mymapping<K,V> {
  function put( key : K, value : V) {...}
```

```
  function get( key : K) : V {...}
}
```

Now you can use this class with strongly typed results:

```
myMap = new Mymapping<String, Integer>
myMap.put("ABC", 29)

theValue = myMap.get("ABC")
```

The `theValue` variable is strongly typed at compile time as `Integer`, not `Object`.

Within the method definition, the values in angle brackets have special meanings as type names in a parameterized class definition. In this case, the `K` and `V` symbols. Use these symbols in method signatures in that class to represent **types** in arguments, return values, and even Gosu code inside the method.

You can think about it as at the time the method runs, the symbols `K` and `V` are *pinned* (assigned) to specific types already. By the time this method runs, some other code created new instances of the parameterized class with specific types already. The compiler can tell which method to call and what types `K` and `V` really represent. In the earlier example, the concrete types are `String` (for `K`) and `Integer` (for `V`).

Gosu generics offer this power to define APIs once and abstract the behavior across multiple types. Define your APIs with the generics and wildcards to generalize your APIs to work with different types of types or collections. Your code is strongly-typed code at compile time, which improves reliability of code at run time.

Parameterized Methods

You can add a finer granularity of type usage by adding the generic type modifiers to the method, immediately after the method name. This is called *parameterizing* the method, or making a *polymorphic method* with a generic type.

For example, in the following example, the class is not parameterized but one method is:

```
package com.example
uses java.util.ArrayList

class Test3
{
  // return the last item in the list
  public function ReturnLast<T>(a : ArrayList<T>) : T{

    var lastItemIndex = a.size - 1
    return a[lastItemIndex]
  }
}
```

Within the method's Gosu code, the symbol `T` can be used as a type and this code works automatically, matching `T` to the type of the collection passed into it.

Code can use this class:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "123"}

var t = new com.example.Test3()

var last = t.ReturnLast( myStrings )

print("last item is: " + last)
```

The variable `last` is strongly typed as `String`, not `Object`.

Other Unbounded Generics Wildcards

In some cases, there is no prior reference to a type wildcard character (such as `T` in earlier examples) if you need to define arguments to a method. This is typical for defining blocks, which are anonymous functions defined in-line within another function (see "Gosu Blocks", on page 213). In such cases, you can simply use the question mark character instead of a letter:

```
var getFirstItem = \ aList : List<?> -> aList[0]
```

For more details about how generics interact with blocks, see "Generics and Blocks" on page 226.

Bounded Wildcards

You can specify advanced types of wildcards if you want to define arguments that work with many types of collections. However, you can still make some types of assumptions about the object's type. For example, you might want to support *homogenous collections* (all items are of the same type) or perhaps only instances of a class and its subclasses or subinterfaces.

Suppose you had a custom class `Shape`. Suppose you want a method to work with collections of circle shapes or collections of line shapes, where both `Circle` and `Line` classes extend the `Shape` class. For the sake of this example, assume the collections are always homogenous and never contain a mix of both types.

It might seem like you could define a function like this:

```
public function DrawMe (circleArray : ArrayList<Shape>)
```

The function would work if you pass it an object of type `ArrayList<Shape>`. However, it would not work if you tried to pass it an `ArrayList<Circle>`, even though `Circle` is a subclass of `Shape`.

Instead, specify support of multiple types of collections while limiting support only to certain types and types that extend those types. Use the syntax "`extends` *`TYPE`*" after the wildcard character, such as:

```
<T extends TYPE>
```

or...

```
<? extends TYPE>
```

For example:

```
public function DrawMe (circleArray : ArrayList<T extends Shape>)
```

In English, you can read that argument definition as "the parameter `circleArray` is an `ArrayList` containing objects all of the same type, and that type extends the `Shape` class".

Although Gosu generics work very similar to generics in the Java language, one other type of bounded wildcard supported by Java is not supported in Gosu. The supertype bounded wildcard syntax `<? super` *`TYPE`*`>` is supported by Java but not by Gosu.

WARNING Gosu does **not** support the generics syntax for bounded supertypes `<? super` *`TYPE`*`>`, which is supported by Java. That syntax is rarely used anyway because the `<? extends` *`TYPE`*`>` is more appropriate for typical code.

Generics and Blocks

The Gosu generics feature is often used in conjunction with another Gosu feature called blocks, which are anonymous functions that can be passed around as objects to other functions. You can use generics to describe or use blocks in two basic ways.

Blocks Can Have Arguments Defined With Generics

You can create a block with arguments and return values that work like the earlier-described function definitions defined with generics. Your block can support multiple types of collections and return the same type of collection passed into it. Use a question mark (`?`) wildcard symbol to represent the type, such as `ArrayList<?>`.

Note: In block definitions you cannot use a letter as a wildcard symbol, such as `ArrayList<T>`. Gosu only supports the letter syntax for parameterized classes and methods.

The following example uses the `<?>` syntax to define an `ArrayList` using generics:

```
uses java.util.ArrayList

// set up some sample data in a string list
```

```
var s = new ArrayList<String>() {"one", "two", "three" }

// define a block that gets the first item from a list
var getFirstItem = \ aList : List<?> -> aList[0]

// call your block. notice that the variable is strongly typed as String, not as Object
var first = getFirstItem(s)

print(first)
```

This code prints the value:

```
one
```

Notice that the return result is strongly typed and Gosu infers the appropriate type from the block.

Functions that Take Blocks as Arguments

Also, there is a more complex type of interaction between blocks and generics. You can pass blocks as objects to other functions. If a function takes a **block as an argument**, you can define that function argument using generics to abstractly describe the appropriate set of acceptable blocks.

To answer questions like “what kind of block does this function support?”, determine the number of arguments, the argument types, and the return type. For example, consider a block that takes a `String` and returns another `String`. The type definition of the block itself indicates one argument, the parameter type `String`, and the return type `String`.

If you want to support a wide variety of types or collections of various types, define the block using generics. If you define your APIs this way, you permit consumers of your APIs to it with a wide variety of types and use strong typing and type inference.

If a class method on a parameterized class (a class using generics) takes a block as an argument, Gosu uses the types of the arguments. You can **omit** the type of the arguments as you define the block.

A typical example of this is the list method `sortBy`, which takes a block. That block takes exactly one argument, which must be the same type as the items in the list. For example, if the list is `ArrayList<String>`, the block must be a `String`. The method is defined as an *enhancement* with the following signature:

```
enhancement GWBaseListEnhancement<T> : java.util.List<T>
...
  public function sortBy( value(T):Comparable ) : java.util.List<T>
```

Note the use of the letter T in the enhancement definition and in the method signature:

```
value(T):Comparable
```

That syntax means that the argument is a block that takes one argument of type T and returns a Comparable value (such as an `Integer` or `String`).

Suppose you had an array list of strings:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}
```

You could easily resort the list based on the length of the strings using blocks. Create a block that takes a `String` and returns the sort key, in this case the text length. Let the `List.sortBy(...)` method handle the rest of the sorting algorithm details and return the new sorted array.

```
var resortedStrings = myStrings.sortBy( \ str -> str.Length() as Integer )
```

It is important to understand that this example omitted the type of the block argument `str`. You do not have to type:

```
var resortedStrings = myStrings.sortBy( \ str : String -> str.Length() as Integer )
```

Type inference in cases like this valuable for easy-to-understand and concise Gosu code that uses generics.

IMPORTANT If you define a block as an argument to a method, you can omit the argument types in the block in some cases. Omit the type if Gosu can infer the type from the arguments required of that method. Omitting the type in cases in which you can do so leads to concise easy-to-read code.

Practical examples of this approach, including the method definitions of the built-in `sortBy` method are shown in the following section, “How Generics Help Define Collection APIs” on page 228.

For extensive information about similar APIs with blocks, see “Gosu Blocks”, on page 213. For specific examples of built-in APIs that use generics with blocks, see “Collections”, on page 231.

How Generics Help Define Collection APIs

By using Gosu generics to define function parameters, you can enforce type safety yet make logical assumptions about interaction between different APIs. This is most notable the Gosu feature called *blocks*, which allows in-line creation of anonymous functions that you can pass to other APIs.

For example, you could easily resort a list of `String` objects based on the length of the strings using these two features combined:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}
var resortedStrings = myStrings.sortBy( \ str -> str.length as Integer)
```

If you want to print the contents, you could print them with:

```
resortedStrings.each( \ str  -> print( str )  )
```

...which would produce the output:

```
a
ab
abc
abcd
```

This concise syntax is possible because the `sortBy` method is defined a single time with Gosu generics.

It uses the wildcard features of Gosu generics to work with **all** lists of type `T`, where `T` could be any type of object, not just built-in types. The method is defined as a Gosu enhancement to all `List` objects. This means that the method automatically works with all Java objects of that class from Gosu code, although the method is not defined in Java. Enhancement definitions look similar to classes. The enhancement for the `sortBy` method looks like:

```
enhancement GWBaseListEnhancement<T> : java.util.List<T>
  ...
  ...
  public function sortBy( value(T):Comparable ) : java.util.List<T> {
    ...
  }
}
```

That means that it works with all lists of type `T`, and the symbol `T` is treated as the type of the collection. Consequently, the `sortBy` method uses the type of collection (in the earlier example, an array list of `String` objects). If the collection is a list of `String` objects, method must takes a comparison function (a *block*) that takes a `String` object as an argument and returns a `Comparable` object. The symbol `T` is used again in the return result, which is a list that has the same type passed into it.

> **IMPORTANT** For a reference of extremely powerful collection-related APIs that use blocks and Gosu generics, see “Collections”, on page 231

Multiple Dimensionality Generics

Typical use of generics is with one-dimensional objects, such as lists of a certain type of object, such as a list of `String` objects, or a collection of `Address` objects. However, generics are flexible in Gosu as well as Java to include multiple dimensionality.

For example, a `Map` stores a set of key/value pairs. Depending on what kind of information you are storing in the `Map`, it may be useful to define APIs that work with certain types of maps. For example, maps that have keys that have type `Long`, and values that have type `String`. In some sense, a `Map` is a two-dimensional collection, and you can define a map to have a specific type:

```
Map<Long, String> contacts = new HashMap<Long, String>()
```

Suppose you want to define an API that worked with multiple types of maps. However, the API would return a value from the map and it would be ideal if the return value was strongly typed based on the type of the map. You could use a 2-dimensional generics with wildcards, to define the method signature:

```
public function GetHighestValue( themap : Map<K,V>) : V
```

The argument `themap` has type `Map` and specifies two type wildcards (single capital letter) separated by commas. In this case, assume the first one represents the type of the key (K) and the second one represents the type of the value (`V`). Because it uses the `V` again in the return value type, the Gosu compiler makes assumptions about relationships between the type of map passed in and the return value.

For example, suppose you pass the earlier example map of type `<Long, String>` to this API. The compiler knows that the method returns a `String` value. It knows this because of the two uses of `V` in the method signature, both as parameter and as return value.

Generics With Custom 'Containers'

Although Gosu generics are most useful with collections and lists, there is no requirement to use these features with built-in `Collection` and `List` classes. Anything that metaphorically represents a "container" for other objects might be appropriate for using Gosu generics to define the type of items in the container.

Abstract Example

Suppose you want to write something that stores key-value maps. Instead of writing:

```
class Mymapping {
  function put( key : Object, value : Object) {...}
  function get( key : Object) : Object {...}
}
```

...you could use generics to define it as:

```
class Mymapping<K,V> {
  function put( key : K, value : V) {...}
  function get( key : K) : V {...}
}
```

Now you can use this class with strongly typed results:

```
myMap = new Mymapping<String, Integer>
myMap.put("ABC", 29)

theValue = myMap.get("ABC")
```

The `theValue` variable is strongly typed at compile time as `Integer`.

Real-world Example

Suppose you were writing a program for an automotive manufacturing company and want to track vehicles within different factories during production. Suppose you want to represent cars with a `Car` object, trucks with a `Truck` object, vans with a `Van` object, and these all derive from a root class `Vehicle`.

You could create some sort of custom container object called `Factory` that does not derive from the built-in collection classes. For the purpose of this example, assume that each factory only contains one type of vehicle. A `FactoryGroup` could contain multiple `Car` objects, or multiple `Truck` objects, or multiple `Van` objects.

Suppose you need APIs to work with all of the following types:

- a `FactoryGroup` containing one or more `Car` objects
- a `FactoryGroup` containing one or more `Truck` objects
- a `FactoryGroup` containing one or more `Van` objects

You could represent these types of collections using the syntax:

- `FactoryGroup<Car>`
- `FactoryGroup<Truck>`
- `FactoryGroup<Van>`

Perhaps you want an API that returns all vehicles in the last step in a multi-step manufacturing process. You could define the API could be defined as:

```
public function GetLastStepVehicles(groupofvehicles FactoryGroup<T>) : FactoryGroup<T>
```

Because the method uses generics, it works with all types of `FactoryGroup` objects. Because both the same letter `T` appears more than once in the method signature, this defines parallelism that tells Gosu about relationships between arguments and/or return values.

The definition of this method could be understood in English as:

> "The method `GetLastStepVehicles` takes one argument that is a factory group containing any one vehicle type. It returns another factory group that is guaranteed to contain the identical type of vehicles as passed into the method."

Alternatively, you could define your API with bounded wildcards for the type:

```
public function GetLastStepVehicles(groupofvehicles FactoryGroup<? extends Vehicle>) : FactoryGroup<T>
```

Using this approach might allow your code to make more assumptions about the type of objects in the collection. It also prevents some coding errors, such as accidentally passing `FactoryGroup<String>` or `FactoryGroup<Integer>`, which fail at compile time. You can find out about your coding errors quickly.

If you want to make code like this, you also need to tell the Gosu compiler that your class is a container class that supports generics. Simply add the bracket notation in the definition of the class, and use a capital letter to represent the type of the class. For example, instead of typing:

```
public class MyFactory
```

...you would instead define your class as a container class supporting generics using the syntax:

```
public class MyFactory<T>
```

Generics with Non-Containers

There is no technical requirement that you use generics with collections or other containers. However, collections and other containers are the typical uses of generics. You can define any class to use Gosu generics to generalize what it supports or how to work with various types. There is no limit on how you can use generics features for new classes.

For example, suppose you want to generalize a class `MyClass` to work differently with different types.

Do not simply define the class `MyClass` as:

```
public class MyClass
```

Instead, define it as:

```
public class MyClass<T>
```

You also could let your class support multiple dimensions similar to how the `Map` class works with two dimentions. See "Multiple Dimensionality Generics" on page 228. You could define your class abstracted across multiple types, separated by commas:

```
public class MyClass<K, V>
```

chapter 17

Collections

The collection and list classes used frequently in Gosu rely on the Java language's collection classes. However, there are important differences because of built-in *enhancements* to these classes that use *Gosu blocks*, anonymous in-line defined functions that are not directly supported in the Java language. Combining Gosu's enhancement and block features permits concise easy-to-understand Gosu code that manipulates collections. With a single line of code, you can loop across collection items and perform actions on each items, extract information from each item, or sort items.

Related topics:

- "Gosu Blocks", on page 213.
- "Gosu Generics", on page 221.
- "Enhancements", on page 209.

IMPORTANT This topic assumes understanding of blocks and generics. To understand how generics enable the enhancement-based APIs, see "How Generics Help Define Collection APIs" on page 228.

This topic includes:

Basic Lists

Lists in Gosu inherit from the Java interface `java.util.List` and its common subclasses such `java.util.ArrayList`.

Creating a List

To create a list with nothing it, specify the type of object it contains in brackets using *generics notation*, such as in this example using an `ArrayList` of `String` objects:

```
var myemptylist = new ArrayList<String>()
```

For more information about generics, see “Gosu Generics”, on page 221.

In many cases you might want to initialize (load) it with data. Gosu has special features that allow a natural syntax for initializing lists similar to initializing arrays.

For example, the following is an example simple array initializer:

```
var s2 = new String[ ] {"This", "is", "a", "test."}
```

In comparison, the following is an example new `ArrayList`:

```
var strs = new ArrayList<String>(){"a", "ab", "abc"}
```

The previous line is effectively shorthand for the following code:

```
var strs = new ArrayList<String>()
strs.add("a")
strs.add("ab")
strs.add("abc")
```

Type Inference and List Initialization

Because of Gosu’s intelligent *type inference*, you can use an even more concise initializer syntax for lists:

```
var s3 = {"a", "ab", "abc"}
```

The type of `s3` is `java.util.ArrayList<String>` (a list of String objects) because all list members have the type `String`.

Gosu infers the type of the `List` to be the *least upper bound* of the components of the list. In the simple case above, the type of the variable `x` at compile time is `List<String>`. If you pass different types of objects, Gosu finds the most specific type that includes all of the items in the list.

If the types implement interfaces, Gosu attempts to preserve commonality of interface support in the list type. This ensures your list acts as expected with APIs that rely on support for the interface. In some cases, the resulting type is a *compound type*, which combines a *class* and one or more *interfaces* into a single type. For example, the following code initializes an `int` and a `double`:

```
var s  = {0, 3.4}
```

The resulting type of `s` is `ArrayList<java.lang.Comparable & java.lang.Number>`. This means that it is an array list of the compound type of the class `Number` and the interface `Comparable`.

> **Note:** The `Number` class does not implement the interface `Comparable`. If it did, then the type of `s` would simply be `ArrayList<java.lang.Number>`. However, since it does not implement that interface, but both `int` and `double` implement that interface, Gosu assigns the compound type that includes the interfaces that they have in common.

Also see related section “Compound Types” on page 308.

Getting and Setting List Values

The following verbose code sets and gets `String` values from a list using the native Java `ArrayList` class:

```
var strs = new ArrayList<String>(){"a", "ab", "abc"}
strs.set(0, "b")
var firstStr = strs.get(0)
```

You can write this in Gosu instead in the more natural index syntax using Gosu shortcuts:

```
var strs = {"a", "ab", "abc"}
strs[0] = "b"
var firstStr = strs[0]
```

Gosu does not automatically resize lists using this syntax. If a list has only three items, the following code does not work:

```
strs[3] = "b" // index number 2 is the higest supported number
```

Gosu provides additional initializer syntax for both lists and maps similar to Gosu's compact initializer syntax for arrays.

Special Behavior of 'List' in Gosu

In `new` expressions, you can use the **interface** type `List` rather than the **class** type `ArrayList`. Gosu treats this special case as an attempt to initialize an instance of the class type `ArrayList`.

For example:

```
var strs = new List<String>(){"a", "ab", "abc"}
```

Basic HashMaps

Maps in Gosu inherit from the Java class `java.util.HashMap`.

Creating a HashMap

To create an empty map, specify the type of objects it contains in brackets using *generics notation*. For example, define a `HashMap` that maps a `String` object to another `String` object:

```
var emptyMap = new HashMap<String, String>()
```

For more information about generics, see "Gosu Generics", on page 221.

In many cases you might want to initialize (load) it with data. Gosu has special features that allow a natural syntax for initializing maps similar to initializing arrays and lists.

For example, the following code creates a new `HashMap` where `"a"` and `"c"` are keys, whose values are `"b"` and `"d"` respectively

```
var strMap = new HashMap<String, String>(){"a" -> "b", "c" -> "d"}
```

That is effectively shorthand for the following code:

```
var strs = new HashMap<String, String>()
strs.put("a", "b")
strs.put("c", "d")
```

This syntax makes it easy to declare static final data structures of this type within Gosu, and with easier-to-read code than the equivalent code would be in Java.

Getting and Setting Values in a HashMap

The following code sets and gets `String` values from a `HashMap`:

```
var strs = new HashMap<String, String>(){"a" -> "b", "c" -> "d"}
strs.put("e", "f")
var valueForE = strs.get("e")
```

You can write this instead in the more natural index syntax using Gosu shortcuts:

```
var strs = new HashMap<String, String>(){"a" -> "b", "c" -> "d"}
strs["e"] = "f"
var valueForE = strs["e"]
```

Creating a HashMap with Type Inference

Because of Gosu's intelligent *type inference* features, you can optionally use a more concise initializer syntax if Gosu can *infer* the type of the map.

For example, suppose you create a custom function `printMap` defined as:

```
function printMap( strMap : Map<String, String> ) {
   for( key in strMap.keys ) {
     print( "key : " + key + ", value : " + strMap[key] )
```

```
    {
  }
```

Because the type of the map is explicit in the function, callers of this function can use an initializer expression **without** specifying the type name or even the keyword `new`. This does not mean that the list is untyped. The list is statically typed but it is optional to declare explicitly because it is redundant.

For example, you could initialize a `java.util.Map` and call this function with verbose code like:

```
printMap( new Map<String, String>() {"a" -> "b", "c" -> "d"} )
```

Instead, simply type the following code and use type inference for concise code:

```
printMap( {"a" -> "b", "c" -> "d"} )
```

Gosu permits this last example as valid and typesafe. Gosu infers the type of the `List` to be the *least upper bound* of the components of the list. In the simple case above, the type of the variable `x` at compile time is `List<String>`. If you pass different types of objects, Gosu finds the most specific type that includes all of the items in the list.

If the types implement interfaces, Gosu attempts to preserve commonality of interface support in the list type. This ensures your list acts as expected with APIs that rely on support for the interface. In some cases, the resulting type is a *compound type*, which combines a *class* and one or more *interfaces* into a single type. For example, the following code initializes an `int` and a `double`:

```
var s  = {"hello" -> 0, "there" -> 3.4}
```

The resulting type of `s` is `HashMap<String, java.lang.Comparable & java.lang.Number>`. This means that it is a map with two generic parameters:

- `String`
- The compound type of the class `Number` and the interface `Comparable`.

Note: The `Number` class does not implement the interface `Comparable`. If it did, then the type of `s` would simply be `Map<String, java.lang.Number>`. However, since it does not implement that interface, but both `int` and `double` implement that interface, Gosu assigns the compound type that includes the interfaces that they have in common.

Special Enhancements on Maps

Just as most methods for lists are defined as part of Java's class `java.util.ArrayList`, most of the behavior of maps in Gosu inherit behavior from `java.util.Map`. However, Gosu provides additional enhancements to extend maps with additional features, some of which use Gosu blocks.

Map Properties for Keys and Values

Enhancements to the `Map` class add two new read-only properties:

- `keys` - calcuates and returns the set of keys in the `Map`. This is simply a wrapper for the keySet() method.
- `values` - returns the values of the `Map`.

Each Key and Value

Enhancements to the `Map` class add the `eachKeyAndValue` method, which takes a block that has two arguments: of the key type and one of the value type. This method calls this block with **each** key/value pair in the `Map`, allowing for a more natural iteration over the Map.

For example:

```
var strMap = new HashMap<String, String>(){"a" -> "b", "c" -> "d"}
strMap.eachKeyAndValue( \ key, value -> print("key : " + key + ", value : " + value ) )
```

List and Array Expansion (*.)

Gosu includes a special operator for array expansion and list expansion. This array and list expansion can be useful and powerful. The expansion operator is an asterisk followed by a period, for example:

```
names*.Length
```

The return value is as follows:

- If you use it on an array, the expansion operator gets a property from every item in the array and returns all instances of that property in a new array.
- If you use it on a list, the expansion operator gets a property from every item in the list and returns all instances of that property in a new list.

For example, suppose you have an array of `Book` objects, each of which has a `String` property `Name`. You could use array expansion to extract the `Name` property from each item in the array. Array expansion creates a new array containing just the `Name` properties of all books in the array.

If a variable named `myArrayOfBooks` holds your array, use the following code to extract the `Name` properties:

```
var nameArray = myArrayOfBooks.Name
```

The `nameArray` variable contains an array whose length is exactly the same as the length of `myArrayofBooks`. The first item is the value `myArrayofBooks[0].Name`, the second item is the value of `myArrayofBooks[1].Name`, and so on.

For another example, suppose you wanted to get a list of the groups a user belongs to so you can display the display names of each group. Suppose a `User` object contains a `MemberGroups` property that returns a read-only array of groups that the user belongs to. In other words, the Gosu syntax `user.MemberGroups` returns an array of `Group` objects, each one of which has a `DisplayName` property. If you want to get the display names from each group, use the following Gosu code

```
user.MemberGroups*.DisplayName
```

Because `MemberGroups` is an array, Gosu expands the array by the `DisplayName` property on the `Group` component type. The result is an array of the names of all the Groups to which the user belongs. The type is `String[]`.

The result might look like the following:

```
["GroupName1", "GroupName2", "GroupName14", "GroupName22"]
```

The expansion operator works with methods also. Gosu uses the type that the method returns to determine how to expand it:

- if the original object is an array, Gosu creates an expanded array
- if the original method is a list, Gosu creates an expanded list

The following example calls a method on the `String` component of the `List` of `String` objects. It generates the list of initials, in other words the first character in each word.

```
var s = {"Fred", "Garvin"}

// get the character array [F, G]
var charArray = s*.charAt( 0 )
```

Array expansion is valuable if you need a single one-dimensional array or list through which you can iterate. Also, there are various enhancement methods for manipulating arrays and lists. For details, see “Enhancement Reference for Collections and Related Types” on page 236.

Important notes about the expansion operator:

- The generated array or list itself is always **read-only** from Gosu. You can never assign values to elements within the array, such as setting `nameArray[0]`.
- The expansion operator `*.` works only for array expansion, never standard property accessing.
- When using the `*.` expansion operator, only **component type properties** are accessible.
- When using the `*.` expansion operator, **array properties** are never accessible.

- The expansion operator applies not only to arrays, but to any `Iterable` type and all `Iterator` types and it preserves the type of array/list. For instance, if you apply the *. operator to a `List`, the result is a `List`. Otherwise, the expansion behavior is the same as with arrays.

Array Flattening to Single Dimensional Array

If the property value on the original item returns an array of items, expansion behavior is slightly different. Instead of returning an array of arrays (an array where every item is an array), Gosu returns an array containing all individual elements of all the values in each array.

Some people refer to this approach as *flattening* the array.

To demonstrate this, create the following test Gosu class:

```
package test

class Family {
  var _members : String[] as Members
}
```

Next, paste the following in to the Gosu Tester window

```
uses java.util.Map
uses test.Family

// create objects that each contain a Members property that is an array
var obj1 = new Family() { :Members = {"Peter", "Dave", "Scott"} }
var obj2 = new Family() { :Members = {"Carson", "Gus", "Maureen"} }

// Create a list of objects, each of which has an array property
var familyList : List<Family> = {obj1, obj2}

// List expansion, with FLATTENING of the arrays into a single-dimensional array
var allMembers = familyList*.Members

print(allMembers)
```

This program prints the following single-dimensional array:

```
["Peter", "Dave", "Scott", "Carson", "Gus", "Maureen"]
```

Application-Specific Examples

For ClaimCenter, the following expression produces an array of `Amount` values for each `Transaction` in the claim's array in the `claim.Transactions` property:

```
claim.Transactions*.Amount
```

The following expression produces an array containing `Amount` values for each `LineItem` in each `Transaction`:

```
claim.Transactions.LineItems*.Amount
```

Use array expansion to make a single one-dimensional array through which you can iterate across. For example:

```
var amounts = TransactionSet.Transactions.LineItems*.Amount
for (n in amounts) { print(n) }
```

Enhancement Reference for Collections and Related Types

The collection and list classes used frequently in Gosu rely on the Java language’s collections and lists classes. However, there are important differences because of built-in *enhancements* to these classes. Gosu *enhancements* are additions to a class or other type that directly add Gosu methods and/or properties to the type, without requiring subclassing to make use of the new features. This is especially useful if you extend Java classes that do not support features, such as adding APIs that use *Gosu blocks*.

Combining Gosu enhancements and Gosu blocks permits concise easy-to-understand Gosu code that manipulates collections. With a single line of code, you can loop across collection items to perform actions on each item, extract information from each item, or sort the collection. For example, Gosu adds the methods `map`, `each`, `sortby`, and other methods to classes.

You can view the source code of these utilities in Guidewire Studio. In the **Enhancements** section within package `gw.util.*`, you find `GWBaseListEnhancement`.

The following table lists some of the collection enhancements. The letter `T` refers to the type of the collection. The syntax `<T>` relates to the feature *Gosu generics*, discussed in "Gosu Generics", on page 221. For example, suppose the argument is listed as:

```
conditionBlock(T):Boolean
```

This means the argument is a block. That block must take exactly one argument of the list's type (`T`) and returns a `Boolean`. Similarly, where the letter `Q` occurs, this represents another type. The text at the beginning (in that example, `conditionBlock` is a parameter that is a block and its name describes the block's purpose.

> **Note:** If a type letter wildcard like `T` or `Q` appears more than once in arguments or return result, it must represent the *same type* each time that letter is used.

Collections Enhancement Methods

Gosu contains enhancement methods for Java collection-related types.

Enhancement Methods on Iterable<T>

Iterable objects (objects that implement `Iterable<T>`) have additional methods described in the following table.

Method/Property	Description
`Count`	Returns the number of elements in the `Iterable`
`single()`	If there is only one element in the Iterable, that value is returned. Otherwise an `IllegalStateException` is thrown.
`toCollection()`	If this Iterable is already of type `Collection`, return it. Otherwise, copy all values out of this Iterable into a new Collection.

Enhancement Methods on Collection<T>

Most collection methods are now implemented directly on `Collection` (not `List` or other similar objects as in previous releases). The following table lists the available methods.

Method/Property Name	Description
`allMatch(cond)`	Returns true if all elements in the `Collection` satisfy the condition
`hasMatch(cond)`	Returns true if this `Collection` has any elements in it that match the given block
`asIterable()`	Returns this `Collection<T>` as a pure `Iterable<T>` (in other words, not as a `List<T>`).
`average( selector )`	Returns the average of the numeric values selected from the `Collection<T>`
`countWhere(cond)`	Returns the number of elements in the `Collection` that match the given condition
`HasElements`	Returns `true` if this `Collection` has any elements in it. This is a better method to use than the default collection method `empty()` because `HasElements` interacts better with `null` values. For example, the expression `col.HasElements()` returns a non-true value even if the expression `col` is `null`.
`first()`	Returns first element in the `Collection`, or return `null` if the collection is empty.
`firstWhere(cond)`	Returns first element in the `Collection` that satisfies the condition, or throws an exception if none do.
`flatMap(proj)`	Maps each element of the `Collection` to a `Collection` of values and then flattens them into a single `List`.
`fold()`	Accumulates the values of an `Collection<T>` into a single `T`.
`intersect(iter)`	Returns a `Set<T>` that is the intersection of the two Collection objects.
`last()`	Returns last element in the `Collection` or return `null` if the list is empty.

Method/Property Name	Description
`lastWhere(cond)`	Returns last element in the `Collection` that matches the given condition, or `null` if no elements match it.
`map(proj)`	Returns a `List` of each element of the `Collection<T>` mapped to a new value.
`max(proj)`	Returns maximum of the selected values from `Collection<T>`
`min(proj)`	Returns minimum of the selected values from `Collection<T>`
`orderBy(proj)`	Returns a new `List<T>` ordered by a block that you provide. Note that this is different than `sortBy()`, which is retained on `List<T>` and which sorts in place. **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.
`orderByDescending(proj)`	Returns a new `List<T>` reverse ordered by the given value. Note that this is different than `sortByDescending()`, which is retained on `List<T>` and which sorts in place. **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.
`partition(proj)`	Partitions this `Collection` into a Map of keys to a list of elements in this `Collection`.
`partitionUniquely(proj)`	Partitions this `Collection` into a Map of keys to elements in this `Collection`. Throws an `IllegalStateException` if more than one element maps to the same key.
`reduce(init, reducer)`	Accumulates the values of a `Collection<T>` into a single V given an initial seed value.
`reverse()`	Reverses the collection as a `List`.
`singleWhere(cond)`	If there is only one element in the `Collection` that matches the given condition, it is returned. Otherwise an `IllegalStateException` is thrown
`sum(proj)`	Returns the sum of the numeric values selected from the `Collection<T>`
`thenBy(proj)`	Additionally orders a `List` that has already been ordered by `orderBy`. **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.
`thenByDescending(proj)`	Additionally reverse orders a `List` that has already been ordered by `orderBy`. **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.
`toList()`	If this `Collection` is already a list, simply return it. Otherwise create a new `List` and copy this `Collection` to it.
`toTypedArray()`	Converts this `Collection<T>` into an array `T[]`.
`union(col)`	Returns a new `Set<T>` that is the union of the two Collections
`where(cond)`	Returns all elements in this Iterable that satisfy the given condition
`whereTypeIs(Type)`	Returns a new `List<T>` of all elements that are of the given type
`disjunction()`	Returns a new `Set<T>` that is the set disjunction of this collection and the other collection
`each()`	iterates each element of the `Collection`
`eachWithIndex()`	Iterates each element of the `Collection` with an index
`join`	joins all elements together as a string with a delimiter
`minBy()`	Returns the minimum T of the `Collection` based on the projection to a `Comparable` object
`maxBy()`	Returns the maximum T of the `Collection` based on the projection to a `Comparable` object
`removeWhere()`	Removes all elements that satisfy the given criteria
`retainWhere()`	Removes all elements that do not satisfy the given criteria. This method returns no value, so it cannot be chained in series. This is to make clear that the mutation is happening in place, rather than a new collection created with offending elements removed.
`subtract()`	Returns a new `Set<T>` that is the set subtraction of the other collection from this collection
`toSet()`	Converts the `Collection` to a `Set`

Methods on List<T>

The following table lists the available methods on `List<T>`.

Method/Property Name	Description
`reverse()`	Reverses the Iterable.
`copy()`	Creates a copy of the list
`freeze()`	Returns a new unmodifiable version of the list
`shuffle()`	Shuffles the list in place
`sort()`	Sorts the list in place
`sortBy()`	Sorts the list in place in ascending order **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.
`sortByDescending()`	Sorts the list in place in descending order. **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.

Methods on Set<T>

The following table lists the available methods on `Set<T>`.

Method/Property Name	Description
`copy()`	Creates a copy of the set
`powerSet()`	Returns the power set of the set
`freeze()`	Returns a new unmodifiable version of the set

The following subsections describe the most common uses of these collection enhancement methods.

Finding Data in Collections

You probably frequently need to find an items in a list based on certain criteria. Use the `firstWhere` or `where` methods in such cases. These functions can be very processor intensive, so be careful how you use them. Consider whether other approaches may be better, testing your code as appropriate.

The `where` method takes a block that returns `true` or `false` and return all elements for which the block returns `true`. The following demonstrates this method:

```
var strs = new ArrayList<String>(){"a", "ab", "abc"}
var longerStrings = strs.where( \ str -> str.length >= 2 )
```

The value of `longerStrings` is `{"ab", "abc"}`. The expression `str.length >= 2` is `true` for both of them.

The `firstWhere` method takes a block that returns `true` or `false` and return the first elements for which the block returns `true`. The following example demonstrates how to find the first item that matches the criteria:

```
var strs = new ArrayList<String>(){"a", "ab", "abc"}
var firstLongerStr = strs.firstWhere( \ str -> str.length >= 2 )
```

The value of `firstLongerStr` is `"ab"`, since `"ab"` is the first element in the list for which `str.length >= 2` evaluates as `true`.

> **WARNING** The find-related list methods simply iterate over the list and can be processor intensive. Depending on the context of your code, a Gosu `find` query may be a better way to compute these values. For more information about `find` queries, see "Find Expressions", on page 159.

Sorting Collections

Suppose you had an array list of strings:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}
```

You can easily resort the list by the length of the `String` values using blocks. Create a block that takes a `String` and returns the sort key, which in this case is the number of characters of the parameter. Let the `List.sortBy(...)` method handle the rest of the details of the sorting and return the new sorted array as the result.

```
var resortedStrings = myStrings.sortBy( \ str -> str.Length )
```

If you want to print the contents, you could print them with:

```
resortedStrings.each( \ str  -> print( str )  )
```

...which would produce the output:

```
a
ab
abc
abcd
```

Similarly, you can use the `sortByDescending` function, which is the same except that it sorts in the opposite order.

For both of these methods, the block must return a comparable value. Comparable values include `Integer`, a `String`, or any other values that can be compared with the ">" or "<" (greater than or less than) operators.

In some cases, comparison among your list objects might be less straightforward. You might require more complex Gosu code to compare two items in the list. In such cases, use the more general sort method simply called `sort`. The `sort` method takes a block that takes two elements and returns `true` if the first element comes before the second, or otherwise returns `false`. The earlier sorting example could be written as:

```
var strs = new ArrayList<String>(){"a", "abc", "ab"}
var sortedStrs = strs.sort( \ str1, str2 -> str1.length < str2.length )
```

Although this method is powerful, in most cases code is more concise and easier to understand if you use the `sortBy` or `sortByDescending` methods instead of the `sort` method.

> **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.

Mapping Data in Collections

Suppose you want Gosu code to take an array list of strings and find the number of characters in each string. Use the list method `map` to create a *new* list where the expression transforms each value and makes the result an element in a new list.

For example:

```
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}
var lengthsOnly = myStrings.map( \ str -> str.length )
```

The value of `lengthsOnly` at the end of this code is an array with elements: 1, 1, 2, 2, 3, 4.

In this example, the `map` method takes a block that is a simple function taking one `String` and returning its length. However, notice that it did not explicitly set the type of the block's argument called `myStrings`. However, this is not an untyped argument, at compile time it is statically typed as a `String` argument. This is implicit because the array list is specified as a list of `String` using the generics syntax `ArrayList<String>`.

Some Gosu collection-related code has concise syntax because collection methods use Gosu *generics*. Generics allow methods such as `map` to naturally define the relationship of types in arguments, return values, and the type of objects in the collection. In this case, the array list is an array list of strings. The `map` method takes a block that **must** have exactly one argument and it **must** be a `String`. Gosu knows the block **must** take a `String` argument so the type can be omitted. Gosu can simply infer the argument type to allow flexible concise code with all the safety of statically-typed code.

The type of the `lengthsOnly` variable also uses type inference and is statically typed. Because the block returns an `int`, the result type of the function must be an `int`. Because of this, `lengthsOnly` is statically typed at compile time to an array of integers even though the type name is not explicit in the code. Specifying the type is optional, and it is good Gosu coding style to use type inference for simple cases like this.

Iterating Across Collections

Now suppose you also want to print each number in the list. You could take advantage of the list method `each`, which can be used in place of a traditional Gosu loop using the `for` keyword:

```
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}
myStrings.map( \ str -> str.length ).each( \ len  -> print( len ) )
```

As you can see, this is a simple and powerful way to do some types of repeated actions with collections. This conciseness can be good or bad, depending on the context of the code. In some cases, it might be better to assign the return value of `map` to a variable and call the `each` method on it. This is especially true if you still need the array of lengths even after printing them. For example:

```
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}
var strLengths = myStrings.map( \ str -> str.length )
strLengths.each( \ len  -> print( len )  )

// maybe use strLengths again in some way here...
```

This is equivalent and some people may find it easier to read.

Partitioning Collections

Blocks are also useful with the `partition` method. This method takes a list and creates a new `java.util.Map` of key/value pairs. The block takes an item from the original list as an argument and returns a value. To perform this task for all input list items, the map keys are results from the block with the input list. Each key points to the input list items that produced that value.

For example, suppose you want take a `String` list and partition it into a `Map` containing the lengths of each `String` value. Suppose the set of input values were the following:

```
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}
```

Each key points to a list of all input `String` values with that length. You could use this one line of Gosu code:

```
var lengthsToStringsMap = myStrings.partition( \ str:String -> str.length )
```

The variable `lengthsToStringsMap` contains a `Map` with four keys:

```
Map { 1 → ["a", "b"], 2 → ["bb", "ab"], 3 → ["abc"], 4 → ["abcd"] }
```

In other words:

- key `1` points to a list of two values, `"a"` and `"b"`
- key `2` points to a list of two values `"bb"` and `"ab"`
- key `3` points to a list with a single value, `"abc"`
- key `4` points to a list with a single value, `"abcd"`

As you can tell from this example, you can make concise and easy-to-read Gosu code with powerful results. Also, note the resulting `Map` is statically typed using type inference.

You can improve your performance if you are sure the output of your block for each list element is always unique. The indirection of having each value wrapped within a list using the `partition` method is unnecessary because there is always a single item in every list. For faster performance in the case in which you know block return results are unique, use the `partitionUniquely` method.

For example:

```
var myStrings = new ArrayList<String>(){"bb", "a", "abcd", "abc"}
var lengthsToStringsMap = myStrings.partitionUniquely( \ str:String -> str.length )
```

The result `Map` has values that are single items not lists:

```
Map { 1 → "a", 2 → "bb", 3 → "abc", 4 → "abcd" }
```

In a real-world situation, you might use code like:

```
//Use a finder to find get a list of claims
var claims = find claim in Claim where ...

//partition the list
var claimsById = claims.partitionUniquely( \ claim -> claim.publicID )
```

The value of `claimsById` is a `Map` of claim `publicID` values to the claims they represent.

If more than one element of the list has the same calculated value for the attribute, the method throws a runtime exception.

Converting Lists, Arrays, and Sets

Use the collection enhancements to convert lists, arrays, and sets as necessary to other types:

- You can convert a `List` or an `Array` to a set by calling `list.toSet()` or `array.toSet()`.
- You can convert a `Set` or an `Array` to a list by calling `set.toList()` or `array.toList()`.
- You can join all of the elements in an `Array` or `List` together with a delimiter by the `join` method, such as:

```
// join all the items in the array together separated by commas
joinedString = array.join(",")
```

Flat Mapping a Series of Collections or Arrays

Use the `flatMap` method to create a single `List` from a series of collections or arrays of objects associated with properties on elements on an outer collection. You provide a block that takes an element of a collection and returns an array or collection, such as an array or collection stored a property of the outer collection. The `flatMap` method concatenates all the items in the returned arrays or collections into a single `List`.

For example, suppose the following structure of your data:

- a claim object has an `Exposures` property that contains an array of exposure objects
- an exposure has a `Notes` property that contains a list of `Note` objects.

First, write a simple block that extracts the note objects from the exposure object:

```
\ e -> e.Notes
```

Next, pass this block to the `flatMap` method to generate a single list of all notes on the claim:

```
var allNotes = myClaim.Exposures.flatMap( \ e -> e.Notes )
```

This generates a single list of notes (on instance of `List<Note>` in generics notation) containing all the notes on all the exposures on the claim.

This method is similar to the Gosu feature called *array expansion* (see “Array Expansion” on page 54). However, it is available on all collections as well as arrays, and `flatMap` method generate different extracted arrays dynamically using a Gosu block that you provide. Your block can perform any arbitrary and potentially-complex calculation during the flat mapping process.

Sizes and Length of Collections and Strings are Equivalent

Gosu adds enhancements for the `Collection` and `String` classes to support both the `length` and `size` properties, so you can use the terms interchangeably with no errors. For collections and strings, `length` and `size` mean the same thing in Gosu.

chapter 18

Gosu and XML

XML files describe complex structured data in a text-based format with strict syntax for easy data interchange. For more information on the Extensible Markup Language, refer to the World Wide Web Consortium web page `http://www.w3.org/XML`. Gosu can read or write to any XML document as a structured tree of untyped nodes. Alternatively, Gosu can parse the XML using an existing XML Schema Definition file (an *XSD file*) to produce a statically-typed tree with structured data. In both cases, Gosu code can interact with XML nodes like native Gosu objects.

This topic includes:

Manipulating XML Overview

Gosu can manipulate structured XML documents in two ways:

- **Untyped nodes.** Any XML can be easily created, manipulated, or searched as a tree of untyped nodes. For those familiar with Document Object Model (DOM), this approach is similar to manipulating DOM untyped nodes. From Gosu, attribute and node values are treated as strings.
- **Strongly typed nodes using an XSD.** If the XML has an XML Schema Definition file (an *XSD file*), you can create, manipulate, or search data with statically-typed nodes that correspond to legal attributes and child elements. If you can provide an XSD file, the XSD approach is much safer. It dramatically reduces errors due to incorrect types or incorrect structure.

Two built-in classes provide the XML features to the Gosu language. The `gw.xml.IXMLnode` interface represents a minimal interface that both typed and untyped XML nodes implement. This allows generic utilities and enhancements that work with both types of XML trees. The `gw.xml.XMLnode` class is the concrete implementation of the `IXMLnode` interface.

The following sections describe the APIs for manipulating XML:

- "Manipulating XML as Untyped Nodes" on page 247
- "Structured XML Using XSDs" on page 251

Not all features of the XSD specification are supported by the statically typed XML features of Gosu. For details, see "Limitations of XSD Support" on page 253.

Exporting XML Data

For all legacy XML node types, you can output the XML node using various properties and methods on the node:

- `node.Bytes` - This property returns an array of bytes (the type `byte[]`) containing the UTF-encoded bytes in the XML. Generally speaking, this is the best approach for serializing an XML document because XML is natively a binary format, not a text format. Compare and contrast with the `asUTFString` method, mentioned later in this list. If your code sends XML with a transport that only understands character (not byte) data, always base-64 encode the bytes to compactly and safely encode binary data. To do this, use the Gosu syntax:

  ```
  var base64String = gw.util.Base64Util.encode(element.bytes())
  ```

  To reverse the process in Gosu, use the code:

  ```
  var bytes = gw.util.Base64Util.decode(base64String)
  ```

  Note: For example, for ClaimCenter messaging, the `Message.Payload` property has type `String` (character data).

- `node.writeTo(java.io.File)` - overwrites the file with the update
- `node.writeTo(java.io.OutputStream)` - writes to a stream but does not close the stream afterward
- `node.asUTFString()` - outputs the node as a `String` value containing XML with a header suitable for later export to UTF-8 or UTF-16 encoding. The generated XML header does **not** specify the encoding. In the absence of a specified encoding, all XML parsers must autodetect the encoding (UTF-8 or UTF-16) based on the byte order mark at the beginning of the document. For more details, refer to:

  ```
  http://www.w3.org/TR/REC-xml/#sec-guessing
  ```

  Although the `asUTFString` method is useful for debugging, generally speaking it is best to use the `node.Bytes` property. Remember to base 64 encode the bytes into a `String` if you use a transport that only understands character data.

  Note: For example, for ClaimCenter messaging, the `Message.Payload` property has type `String` (character data).

  For more information about UTF-8, refer to:

  ```
  http://tools.ietf.org/html/rfc3629
  ```

  WARNING Remember to test that your receiving system to ensure that it properly decodes UTF-8 text . Be careful to test your integration code with non-English characters (characters with high Unicode code points).

For example:

```
var xmlBytes = root.Bytes
var xmlString = root.asUTFString()
```

For Guidewire application messaging, follow the pattern of the following ClaimCenter Event Fired rule code. It creates a new message containing the XML for a contact entity as a `String` to work with the standard message payload property.

```
if (MessageContext.EventName == "ContactChanged") {
  var xml = new mycompany.messaging.ContactModel.Contact( MessageContext.Root as Contact )
  var strContent = gw.util.Base64Util.encode(xml.Bytes)
  var msg = MessageContext.createMessage(strContent)

  print("Message payload of my changed contact for debugging:")
  print(msg);
}
```

Your messaging transport code takes the payload `String` and exports it as UTF-8:

```
override function send( message : Message, transformedPayload : String ) : void  {
var bytes = bgw.util.Base64Util.decode(base64String)

   // send this byte array to a foreign system...

   message.reportAck();
}
```

The examples in this section by default generate the entire XML graph. If you are using a Guidewire Standard XML model in Studio (see “The Guidewire XML (GX) Modeler” on page 262), optionally you can choose other export options, such as:

- You can selectively export elements even for `null` properties using the `Verbose` property in `GXOptions`. For details, see “Customizing Guidewire Model XML Output (GXOptions)” on page 268.
- You can selectively export only entity fields that changed using the `Incremental` property in `GXOptions`. For details, see “Customizing Guidewire Model XML Output (GXOptions)” on page 268.

Manipulating XML as Untyped Nodes

To load XML into Gosu’s native parser, follow these steps:

1. **Get XML as String, File, or InputStream.** Get a reference to XML raw data in the form of a Gosu `String`. Alternatively, you can use Java objects of type `java.io.File` or `java.io.InputStream` if you have code to stream the contents of the XML directly through some other mechanism from Java code.
2. **Parse the XML into a node.** Pass the XML (the `String`, `File`, or `InputStream`) to the `gw.xml.XMLNode.parse(...)` method.
3. **Manipulate XML nodes and values.** Set values, get values, or create new nodes.

For example, to parse a `String` into an `XMLNode`:

```
var xml =
   "<?xml version='1.0' encoding='utf-8'?>" +
   "<slideshow title=\"Sample Slide Show\">" +
   "<slide title=\"My First Slide\">" +
     "<summary>Once upon a time, there was a gastropod.</summary></slide>" +
   "</slideshow>"

var root = gw.xml.XMLNode.parse( xml )
```

Once you have a reference to a node, you can use the untyped node operations described in the following section.

If you want to manipulate XML nodes but start with an empty initial tree rather than based on initial XML, create the initial node with the new element approach. See “New Elements” on page 248 for details.

Untyped Node Operations

Once you have an `XMLNode`, you can perform simple operations on the nodes.

Attribute Values

Get attribute values from a node with the syntax:

```
valueString = node.Attributes[AttributeNameAsString]
```

For example:

```
valueString = node.Attributes["color"]
```

Set attribute values from a node with the syntax:

```
node.Attributes[AttributeNameAsString] = valueString
```

For example:

```
node.Attributes["color"] = "red"
```

New Elements

Create a new element or root node by creating a new `XMLNode` and passing it the element's name in the constructor, as a `String`.

```
var node = new gw.xml.XMLNode(ElementName)
```

For example, create a new `<Name>` element as regular node or a root node using the syntax:

```
uses gw.xml.*

var node = new XMLNode("Name")
```

Add Child Elements to a Parent Node

Add child elements to a parent node using the syntax:

```
parentNode.Children.add(newNode)
newNode.Parent = parentNode
```

The parent property on the child is not automatically set after you add a node to a list of children.

Getting and Setting Text Contents of an Element

To get or set the text contents of an element, simply set the `text` property of the element:

```
var xml =
   "<?xml version='1.0' encoding='utf-8'?>" +
   "<slideshow title=\"Sample Slide Show\">" +
   "<slide title=\"My First Slide\">" +
     "<summary>Once upon a time, there was a gastropod.</summary></slide>" +
   "</slideshow>"

var root = gw.xml.XMLNode.parse( xml )

// getting text contents from an element
var t = root.Children[0].Children[0].text
print ("contents are " + t)

// setting text contents in an element
root.Children[0].Children[0].text = "In the beginning, " + t + " The End."
```

Exporting the XML as a String

Convert an `XMLNode` and all that node's descendents into a `String` representation of the XML with the `Bytes` property (the preferred method) or the `asUTFString` method from any node. Use either API on the root node to export the entire tree. For example:

```
var xmlBytes = root.Bytes
var xmlString = root.asUTFString()
```

For more details of XML export and issues related to Unicode and Guidewire messaging, see "Exporting XML Data" on page 246.

Example of Manipulating XML as Untyped Nodes

The following example demonstrates the following steps:

1. Defines XML as String data.
2. Creates a root node.
3. Gets attribute values from a node.

4. Sets attribute values on a node
5. Gets child nodes from a node.
6. Creates a new element.
7. Create a new child node.
8. Exports the manipulated XML as `String` data.

You may paste this code into the Gosu Tester to manipulate XML data representing a slide show:

```
var xml =
   "<?xml version='1.0' encoding='utf-8'?>" +
   "<slideshow title=\"Sample Slide Show\">" +
   "<slide title=\"My First Slide\">" +
     "<summary>Once upon a time, there was a gastropod.</summary></slide>" +
   "</slideshow>"

var root = gw.xml.XMLNode.parse( xml )

// get a value
print ("slideshow title is " + root.Attributes["title"])

//set values
root.Attributes["title"] = "My NEWER Slideshow Title"
root.Attributes["author"] = "Andy Applegate"
print ("slideshow title is: " + root.Attributes["title"])
print ("author is: " + root.Attributes["author"])

//get a reference to a child node. root.Children is a List
var firstSlide = root.Children[0]
print("slide 0 element name is: " + firstslide.ElementName)
print ("slide 0 title is: " + firstSlide.Attributes["title"])

//get a child of that node...
var firstSlideSummary = firstSlide.Children[0]
print("slide 0's summary text (in child element) is: " + firstslide.Children[0].Text)

// create a new element for another slide
var secondSlide = new gw.xml.XMLNode("slide")
secondSlide.Attributes["title"] = "My Second Slide Title"
root.Children.add( secondSlide )
secondSlide.Parent = root

// create a summary child of the new slide
var secondSlideSummary = new gw.xml.XMLNode("summary")
secondSlideSummary.Text = "The end."
secondSlide.Children.add( secondSlideSummary )
secondSlideSummary.Parent = secondSlide

// Confirm that the new add node was successful by printing all the XML...
xml = root.asUTFString()
print(xml)
```

This program prints the following output:

```
slideshow title is Sample Slide Show

slideshow title is: My NEWER Slideshow Title

author is: Andy Applegate

slide 0 element name is: slide

slide 0 title is: My First Slide

slide 0's summary text (in child element) is: Once upon a time, there was a gastropod.

<slideshow author="Andy Applegate" title="My NEWER Slideshow Title">
  <slide title="My First Slide">
    <summary>Once upon a time, there was a gastropod.</summary>
  </slide>
  <slide title="My Second Slide Title">
    <summary>The end.</summary>
  </slide>
</slideshow>
```

Collection-like Enhancements with XML Nodes

Given an XML node, you can easily extract a list of direct child nodes (child elements), or a list of all nodes (all descendent elements) in the tree. In both cases, Gosu returns the nodes as a `List` of `XMLNode` objects, which is specified in Gosu Generics notation as `List<XMLNode>`. The Gosu language provides built-in enhancements (methods added to classes) for these Java-based collection classes. You can use these collection enhancements to find and iterate across multiple items in a list of XML nodes in Gosu. Additionally, the `XMLNode` class itself provides methods (implemented through enhancements) that mirror several of these helpful utilities directly on the `XMLNode` element.

> **Note:** For more information, see “Gosu Generics”, on page 221. Be aware that in Gosu, the type called `List` is shorthand for the Java utility class `java.util.ArrayList`.

The following three methods exist on the `XMLNode` element itself and also any lists of nodes extracted from a node.

The list method `findAll(...)` accepts an in-line function called a *block* expression, which allows you to concisely describe XPath-style searches. You can use either simple queries or complex Gosu calculations within your block expression. You can use this method directly on `XMLNode` or on collections of `XMLNode` elements.

You can also use `findFirst`, which is similar to `findAll` but only returns the first match of the supplied criteria. You can use this method directly on `XMLNode` or on collections of `XMLNode` elements.

To iterate across all descendent nodes (not just direct child nodes), use the `each` method which takes one parameter that is a block that describes the action to perform.

If you call `each` method or the `findFirst` method directly on a node, the methods operate on tree nodes in *depth-first* order. Depth-first order in this context means that it recursively processes each node before proceeding to the next sibling. Gosu processes each node and then processes each of its child nodes before the system processes a node’s own siblings. For example, suppose a root node had three child nodes and each of those child nodes had five child nodes which are grandchildren of the root node. The `each` method processes the root node first. Next, it processes the first child node. Next, it processes the five grandchildren of the first child. Next, it processes the other two child nodes of the root recursively in the same way.

> **IMPORTANT** For a full reference with examples of collection-related APIs, most of which use blocks, see “Collections”, on page 231. For more information about blocks, see “Gosu Blocks”, on page 213.

To search or iterate across all nodes in the tree or iterate across all items, use collection-like enhancement methods directly on the root node. These methods on the root node refer to **all** nodes in the tree.

To search or iterate across only direct child nodes, use the collection enhancement methods on the list returned on the `Children` property of the desired node. In other words, call methods on the value of `node.Children`. This returns only direct children, and is not recursive.

For example:

- Search **all nodes** in the tree recursively for an element name, similar to XPath expression `"\\MyElement"`, given an existing `XMLNode` assigned to a local variable `root`:
  ```
  root.findAll(\ n -> n.ElementName == "MyElement")
  ```
- Search **direct child nodes** of a the root node for a specific element name:
  ```
  root.Children.findAll(\ n -> n.ElementName == "MyElement")
  ```
- Print the name of **every element** in the tree:
  ```
  root.each(\n -> print(node.ElementName))
  ```
- Search **all nodes** in the tree for an attribute named `"mytitle"` and that has exactly one child element:
  ```
  var titledAndOneChild = root.findAll( \ n -> n.Attributes["mytitle"] != null AND n.Children.size == 1 )
  ```
  Then, to test the results with the previous example, print out each title you find:
  ```
  titledAndOneChild.each(\n -> print("found slide with title: " + n.Attributes["title"]))
  ```

For other examples of collection APIs, see “Collections”, on page 231. However, many of those methods can only be called on actual collections, not a `XMLNode` element. If you want to add additional functionality to the `XMLNode` element, you can add enhancements to the `IXMLNode` interface. Refer to the built-in enhancement `gw.util.xml.GWBaseIXMLNodeEnhancement` to see similar code that you can use to make new enhancements.

IMPORTANT If you call collection-related methods discussed in this section directly on a node, you are calling methods defined **on the node itself**. These include the `findAll`, `findFirst`, and `each` method for each node object. If you reference the direct child nodes with *`node`*`.Children`, Gosu generates a Gosu `List` of nodes (`List<XMLNode>`). On these lists, collection methods available are the larger set of `List` enhancement methods. See “Collections”, on page 231 for more information about these enhancement methods.

Attribute Manipulation as Maps

Gosu stores attributes of an `XMLNode` in maps (instances of `java.util.Map`) of the type `Map<String, String>`. That is generics syntax for a map that maps a `String` value to another `String` value. Use all of the Java language `Map` methods as well as the Gosu `Map` enhancements if manipulating a node’s attributes.

A simple example of this is to set attributes:

```
myNode.Attributes["mytitle"] = "My Slideshow Title"
```

However, you can use more advanced `Map` enhancements, such as the `eachKeyAndValue` method that iterates across all key and value pairs in the map:

```
xml = root.Attributes.eachKeyAndValue( \ s, s2 -> print("I contain key " + s + " and value " + s2 ))
```

For the example in the previous section, this prints the following:

```
I contain the key title and value My NEWER Slideshow Title
I contain the key author and value Andy Applegate
```

Structured XML Using XSDs

Gosu lets you read and write XML trees using *strongly-typed* Gosu objects with properties and subelements that correspond to the legal attributes, child elements, and legal structure of the document. This approach requires an XML Schema Definition file, also known as an *XSD file*. If you provide an XSD file, this approach is much safer than treating the tree as untyped nodes, since it dramatically reduces errors due to incorrect types or incorrect structure.

Note: Contrast this strongly-typed node approach to treating XML data as *untyped* nodes as discussed in see “Manipulating XML as Untyped Nodes” on page 247. The untyped node approach does not require an XSD file.

In explaining how the types are used, it helps to use an example XSD file. This section references the following example XSD file:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:simpleType name="Color">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Red"/>
      <xsd:enumeration value="Blue"/>
      <xsd:enumeration value="Green"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:element name="SimpleTest">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Test1"/>
        <xsd:element ref="Test2" maxOccurs="unbounded"/>
        <xsd:element name="Test3" type="Color"/>
        <xsd:element name="Test4" type="TestType" minOccurs="1" maxOccurs="unbounded"/>
```

```
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="TestType">
    <xsd:attribute name="number" type="xsd:integer"/>
    <xsd:attribute name="color" type="Color"/>
  </xsd:complexType>

  <xsd:element name="Test1" type="TestType"/>

  <xsd:element name="Test2">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Test1"/>
      </xsd:sequence>
      <xsd:attribute name="final" type="xsd:boolean"/>
      <xsd:attribute name="id" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

XSD Location and Generated Types

To tell Gosu to load your XSD, put your XSD files in the same file hierarchy (in the `configuration` module) as Gosu classes. The file hierarchy **location** of your XSD in the class hierarchy defines the package into which Gosu loads your XSD. In other words, where you put the XSD defines the Gosu package for your XSD and all its related objects.

> **IMPORTANT** When creating directories manually, the target location directory might not already exist. You might need to create the directory at your desired location. Remember to add your XSD file to your source code management system.

For example, suppose you add the previous example XSD file at the path:

`ClaimCenter/modules/configuration/gsrc/`**`mycompany/messaging/MyData.xsd`**

The package name for types **generated** from this XSD appears in the package `mycompany.messaging.MyData.*`.

Gosu generates the following classes to represent elements in that example XSD:

```
mycompany.messaging.MyData.SimpleTest
mycompany.messaging.MyData.TestType
mycompany.messaging.MyData.Test2
mycompany.messaging.MyData.Color
```

Gosu generates one enumeration class:

```
mycompany.messaging.MyData.enums.ColorEnum
```

All generated node classes extend the `IXMLNode` interface (in the `gw.xml` package). This means that all methods, properties, or enhancements defined on `IXMLNode` are available on all generated node classes. For example, you can search or iterate use the `findAll` method or `each` method on a node. See “Collection-like Enhancements with XML Nodes” on page 250.

Every class generated for an XSD `simpleType` or `complexType` has a property for each attribute and for each possible child element. If a child element can only occur once, it appears as a property of the appropriate element type. In contrast, a child allowed to appear multiple times appears as a `List` with the appropriate generic type. Classes generated for types that allow a text value also have a `Value` property for accessing the node’s text. If a child element occurs once and only has a text value and no additional attributes or children, Gosu hides the child and exposes the text value as a property. Lastly, all elements have a `asUTFString` method. The `asUTFString` method transforms the element tree into a XML string as well as static `parse()` methods that can transform a `String`, `File`, or `InputStream` into a node tree.

Enumerated type classes function as enumerations in Gosu, with static properties generated for each possible enumeration value.

What Types Does Gosu Generate?

Every `complexType` or `simpleType` declaration in the XSD, whether named or anonymous, causes a new type class. Every `simpleType` that defines an enumeration restriction also defines an enumeration type in the `enums` subpackage, with the suffix `"Enum"`. For instance, in the following example the enumeration restriction on the `Color` type generates the `xsd.enums.ColorEnum` enumeration class.

The names of types match the XSD type's name if it has one, or otherwise the name of the enclosing element if the type is anonymous.

In addition, Gosu generates classes for grouping objects that you cannot in-line:

- Gosu generates classes for what is known in the XSD format as a *choice*, which permits one of several elements to appear. (See "Handling Choices in XML" on page 256 for more information.)
- Gosu generates classes for XSD sequences that appear more than once

Gosu applies more complicated rules for nested in-line type definitions and type name conflicts using a three-step process:

1. Gosu first creates a tentative initial name for the class. If the type is named, Gosu uses the name of the type to define the new class name. If the type is not named, Gosu has to create an in-line definition. If the element is define as part of another type definition, the class name is the enclosing type name, then an underscore character (_), then the element name. For example, if the enclosing type is `MyAddress` and the element name is `MyPhone`, then the new class would be named `MyAddress_MyPhone`. If the element is not part of another type definition, Gosu uses just the element name.
2. Gosu next normalizes the name to conform to certain rules. For example, if the name does not start with a letter or underscore character (_), Gosu prepends an underscore character. Additionally, any characters that are not a letter, digit, or underscore character are converted to an underscore.
3. If the normalized name is already taken, append the number 2 to the name, or 3 if that is taken, and so on. For example, if the name is `MyAddress_MyPhone` but that is already taken then Gosu tries `MyAddress_MyPhone2`. If that is taken, then Gosu tries `MyAddress_MyPhone3`, and so on, until the name is unique.

XSDs that Reference Other XSDs Through HTTP

For maximum performance and reliability, the Gosu XSD type loader does not follow HTTP links. If an XSD references non-local resources such other XSDs accessed from HTTP, place copies of the XSDs in the same directory as your main schema. In other words, put the XSDs directly in the Gosu class hierarchy next to the other XSD. Otherwise, Gosu cannot parse your main XSD. You do **not** need to modify the XSD in any way to support loading the XSD locally. Instead, Gosu looks for a local resource with the same name as that XSD in the same directory.

Limitations of XSD Support

Some features of the XSD specification have special limits:

- `<xsd:List>` always converts to a `String` in Gosu
- `<xsd:Union>` always converts to a `String` in Gosu
- `<xsd:redefine>` is unsupported
- other XSD schema elements that impose restrictions are not restricted from within Gosu. However, if you are importing XML from another source, you can validate the XML against the XSD fully using an optional boolean argument when you parse the XML. For more information, see "Importing Strongly-Typed XML" on page 254.
- The XSD modeler does not support qualified attributes.

If you have questions about the Gosu parsing of a specific XSD, contact Guidewire Professional Services.

Importing Strongly-Typed XML

Parsing (importing) incoming XML documents is fairly simple. To parse XML into the Gosu in-memory graph that represents the XML, call the `parse` method on the root node class that mirrors the incoming document root node. For example:

```
var node = xsd.test.SimpleTest.parse(myXMLstring)
```

Once you have a reference to the root node, use the strongly-typed getter methods to access child elements. For example, use the various property accessors instead, assisted by the Gosu editor code completion features. To get the color of the `Test1` node under the root node, use straightforward strongly-typed code such as:

```
var color = node.Test1.color
```

However, since the strongly typed nodes implement the `IXMLNode` interface, you can use the untyped `Children` and `Attributes` properties. Using these untyped properties treat the tree essentially as a untyped DOM-like tree. For related discussion, see “Manipulating XML as Untyped Nodes” on page 247.

For example, get an attribute with code with code similar to the following:

```
// untyped access of the number attribute
x.Attributes["number"]
```

Similarly, you can use the enhancements discussed in “Collection-like Enhancements with XML Nodes” on page 250.

If you use the untyped node features, including the collection enhancements, you can continue to use the strongly typed node access by coercing nodes to the appropriate node class. For example, suppose you want to find the `Test4` child element that has `number` attribute set to 5 and work with that node as strongly typed. You can use code such as:

```
// Use untyped "Attributes" and "Children" properties to find the nodes you want
var n1 = node.findFirst(\x -> x.ElementName == "Test4" && x.Attributes["number"] == "5")

// coerce to specific node class imported from XSD.
var n2 = n1 as xsd.test.TestType

// (In practice, do the coercion "... as TYPENAME" on the same line as the previous line.)


// Access node subelements and attributes as STRONGLY-typed values...
var color = n2.Test1.color
```

You can combine these features in other ways, for example:

- To create a list of the names of elements that are children of the root node:

```
var elementNames = node.Children.map(\x -> x.ElementName)
```

- To compile a list of all the ids of the `Test2` nodes:

```
var ids = node.Test2s.map(\x -> x.id)
```

If you use an XSD for strongly-typed data access, you can use both the strongly-typed operations and untyped node access, freely intermixing the two types of operations.

Optional Validation During Import

As you import XML, you can validate the XML against the XSD fully using an optional Boolean argument to the `parse` method when you parse the XML. If you pass `true`, Gosu validates the XML against the XSD and throws an exception if it fails.

For example:

```
uses gw.xml.XMLNode

// somehow get a reference to some XML to import. In this example, we use a trivial XSD
// that has no limitations, and just generate XML bytes from it
var UTF_XML = new xsd.test.Root().asUTFString()

// parse it, and pass an optional boolean to the parse() method. It indicates whether to
// validate the incoming XML against the XSD. Thows an exception if there are errors.
var newNode = new xsd.test.Root().parse(UTF_XML, true)
```

Writing Strongly-Typed XML

Constructing an XML tree is more complicated than working with XML that has already been constructed with a parse method call, but the basic idea is relatively simple. First, create a top-level object, then set its properties, attach subobjects, and so on until the object tree is complete. To convert the tree into an XML string, invoke the `asUTFString`() method on the top-level node or get the `Bytes` property from the node.

For more details of XML export and issues related to Unicode and Guidewire messaging, see “Exporting XML Data” on page 246.

To create a node, first use the no-argument constructor for the associated class:

```
var simpleTest = new xsd.test.SimpleTest()
```

To set node properties, use the strongly-typed properties or access attributes using the `Attributes` untyped map:

```
simpleTest.id = "RootNode"              // strongly typed
simpleTest.Attributes["id"] = "RootNode" // untyped
```

To add a single-element child property, simply set the property:

```
simpleTest.Test1 = new xsd.test.TestType()
```

To modify a multiple-element child property, instead add to the list that is always present for that child:

```
simpleTest.Test2s.add(new xsd.test.Test2())
```

You can also assign a new array to the value, but the preferred method is to simply add and remove from the pre-existing list. If you want the list of children to be emptied, call clear on the list rather than setting the list to `null`.

Also, the properties are strongly-typed so they imply both the element names and the ordering of the elements relative to the other properties. Thus, you can create the children elements and attach them in any order, and the resulting XML orders them correctly as per the schema. They appear in output in the natural order in a list of elements and the order is as defined in XSD.

Once the tree is complete, you can produce the resulting XML as a string by calling the `asUTFString()` method or get the `Bytes` property. For example:

```
var xmlBytes = root.Bytes
var xmlString = root.asUTFString()
```

For more details of XML export and issues related to Unicode and Guidewire messaging, see “Exporting XML Data” on page 246.

The following Gosu manipulates XML nodes using the test XSD from earlier in this topic:

```
var simpleTest = new xsd.test.SimpleTest()
simpleTest.id = "Root"

var test2 = new xsd.test.Test2()
test2.id = "test"

simpleTest.test2s.add(test2)
simpleTest.test2s.add(new xsd.test.Test2())
simpleTest.test2s.get(1).final = true
simpleTest.test2s.get(1).Test1 = new xsd.test.TestType()

var test1 = new xsd.test.TestType()
test1.color = Red // Gosu can infer what enum class is appropriate!
test1.number = 5

simpleTest.test4s.add(new xsd.test.TestType())
simpleTest.test3 = Blue // Since this is a simple child element, you access its value directly

print(simpleTest.asUTFString())
```

The final print statement produces this XML tree:

```
<SimpleTest id="Root">
  <Test1 color="Red" number="5"/>
  <Test2 id="test"/>
  <Test2 final="true">
    <Test1/>
  </Test2>
  <Test3>Blue</Test3>
```

```
  <Test4/>
</SimpleTest>
```

Remember that element names are only guaranteed to be assigned at the time the element writes out to XML. Because some types correspond to multiple element names, it is not always possible for Gosu to determine the element name until **after** a node is assigned to a parent. at that point, the XSD's structure defines which element the node represents.

Handling Choices in XML

Gosu specially handles XSD choices are handled specially, as mentioned in "What Types Does Gosu Generate?" on page 253. There are several things to know about XSD choices, which represent the ability for the XML to contain one (and only one) of several different elements to be present.

For example, the following is a simple XSD containing a choice:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="HomePhone" type="xsd:string"/>
  <xsd:element name="WorkPhone" type="xsd:string"/>
  <xsd:element name="MobilePhone" type="xsd:string"/>

  <xsd:element name="LastVerifiedBy" type="xsd:string"/>
  <xsd:element name="LastVerifiedDate" type="xsd:date"/>

  <xsd:element name="PhoneNumber">
    <xsd:complexType>
      <xsd:sequence>
      <xsd:element ref="LastVerifiedBy" minOccurs="0"/>
      <xsd:element ref="LastVerifiedDate" minOccurs="0"/>
       <xsd:choice>
        <xsd:element ref="HomePhone"/>
        <xsd:element ref="WorkPhone"/>
        <xsd:element ref="MobilePhone"/>
      </xsd:choice>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Save this XSD as the file:

```
ClaimCenter/modules/configuration/gsrc/mycompany/messaging/ChoiceTest.xsd
```

From a programming perspective, generally speaking Gosu creates a new object that represents the choice itself and creates subobjects and special properties on the choice group object. If an XSD element has only one choice subelement and the choice is not the element's root, access the choice object through the `choice` property on the parent object.

If there is more than one choice in an element, access additional choices through the properties `property2`, `property3`, and so on. To access individual subobjects for each choice within that choice group, Gosu creates subobjects (and in some cases, other shortcut properties). You can access these subobjects through the choice object by the individual choice's ID. Gosu creates one or two properties for each choice, depending on whether the type is a simple type. If it is a simple type, such as an `XSDString` or `XSDInt`, Gosu encodes it as a text value with no child elements.

If the type of the choice is a complex type, Gosu creates one property on the choice object for the choice:

- The element property's name matches the ID of the choice value, with no suffix. This property sets or gets the element itself. Use the new operator as necessary to create new elements to assign to those properties, such as the following example (does not use the earlier XSD example):

```
var a = new mycompany.messaging.ChoiceTest.MyCustomComplexType()
```

If the type of the choice is a simple type, Gosu creates two properties:

- A shortcut property for each choice with a name matching the ID of the choice value. This property sets or gets the contents of the element. If Gosu can coerce the Gosu data to the appropriate type, Gosu implicitly coerces for set and get operations. In this case, in this case Gosu can coerce a `String` to the value of an

XSDString. With the example XSD, the PhoneNumber choice element has properties HomePhone, WorkPhone, and MobilePhone. Those choices are simple types, specifically the XSDString type.

- An element property with a name matching the ID of the choice value, with the suffix "_elem". This property sets or gets the element itself. Accessing the element directly is most important if you need to get or set attributes on the element itself. With the example XSD, the PhoneNumber choice element as properties HomePhone_elem, WorkPhone_elem, and MobilePhone_elem.

IMPORTANT If the type is a simple type, the type gets a shortcut property to get and set the value and an element property with the _elem suffix. Otherwise, Gosu cannot create the shortcut property to get or set the value, so the element property has the property name with no suffix.

The following code using the example XSD sets the contents of the home phone element with a standard String:

```
var a = new mycompany.messaging.ChoiceTest.PhoneNumber()

// Set the choice value using a special shortcut that creates a choice child element and sets a value
a.choice.HomePhone = "415 555 1212"
```

The following code sets the contents of the home phone element with a standard String. Once the choice's child element exists (and thus is now non-null), use the HomePhone_elem property to access the element directly.

```
var a = new mycompany.messaging.ChoiceTest.PhoneNumber()

print(a.choice.HomePhone_elem)              // print NULL -- no choice is selected, subobjects=null
a.choice.HomePhone = "415 555 1212"         // CREATE the element subproperty. It is now non-null

// Use the new element to access the value property
a.choice.HomePhone_elem.value = "415 867 5309"

// Use the new element to set or get attributes
a.choice.HomePhone_elem.attributes["extension"] = "x12345"

print(a.asUTFString())
```

This code prints:

```
null

<PhoneNumber>
  <HomePhone extension="x12345">415 867 5309</HomePhone>
</PhoneNumber>
```

Since the premise of a choice is that no more than one of the choices can be chosen at any one time, Gosu enforces this requirement. If you set the value of one of the choices, any previous choices are removed. Gosu does this by setting the subelement for that choice to null. If there is a shortcut property for the value of a simple type, getting that property also returns null. In the example XSD, this means that if you set the home phone, then set the work phone, that the HomePhone and HomePhone_elem properties now have the value null.

For example, take the following Gosu code:

```
var a = new mycompany.messaging.ChoiceTest.PhoneNumber()
print(a.choice.HomePhone_elem)     // print NULL -- no choice is selected, subobjects=null

a.Choice.HomePhone = "415 555 1212"
a.Choice.WorkPhone = "415 666 1234"

// the home choice is now null because it was replaced by the workphone
print(a.Choice.HomePhone_elem)

// the home choice element value shortcut also returns null
print(a.Choice.HomePhone)

// the work phone?
print(a.Choice.WorkPhone)
```

It prints the following:

```
null

null

null
```

```
415 666 1234
```

If you get the value and that is not the current selection (not the current choice of the choice group), Gosu always returns `null`. If you are setting the value for the choice and it is not already the current selection, it sets that value as the choice and removes any previous choice.

If a choice has never been for that element yet, the properties for all choice elements have the value `null`.

If a Choice is Root of an Element

There is a special case if a choice element is defined at the root of the element, rather than enclosed in a sequence or other type of structure. If this is the case, you do not need to type the `root.Choice.`*`choicename`* syntax because it is unnecessary to disambiguate the choices from other properties on the element. Instead, the property names appear directly on the root element rather than a separate subobject accessible using the `choice` property.

Using the example XSD, note the definition of the `Email` element has a choice as the root of the element. Consequently, the properties that normally would appear on the `choice` subobject are flattened such as they appear on the root.

For example, do not write this:

```
var a = new mycompany.messaging.ChoiceTest.Email()

a.choice.HomeEmail = "abc@def.com"
print(a.choice.HomeEmail_elem.value)
```

Instead, Gosu creates choice properties directly on the root element, so use code such as the following:

```
var a = new mycompany.messaging.ChoiceTest.Email()

a.HomeEmail = "abc@def.com"
print(a.HomeEmail_elem.value)
```

Gosu Type to XSD Type Conversion Reference

The following table lists conversions the Gosu use to convert from Gosu types to XSD types.

XSD type in custom XSD	Maps to this Gosu type
`xsd:ENTITIES`	`java.lang.String`
`xsd:ENTITY`	`java.lang.String`
`xsd:ID`	`java.lang.String`
`xsd:IDREF`	`gw.xml.xsd.IXMLNodeWithID<gw.xml.IReadOnlyXMLNode>`
`xsd:IDREFS`	`java.util.List<gw.xml.xsd.IXMLNodeWithID<gw.xml.IReadOnlyXMLNode>>`
`xsd:NCName`	`java.lang.String`
`xsd:NMTOKEN`	`java.lang.String`
`xsd:NMTOKENS`	`java.util.List<java.lang.String>`
`xsd:NOTATION`	`java.lang.String`
`xsd:Name`	`java.lang.String`
`xsd:QName`	`javax.xml.namespace.QName`
`xsd:anyType`	`null`
`xsd:anyURI`	`java.net.URI`
`xsd:base64Binary`	`byte[]`
`xsd:boolean`	`java.lang.Boolean`
`xsd:byte`	`java.lang.Byte`
`xsd:date`	`gw.xml.xsd.types.XSDDate`
`xsd:dateTime`	`gw.xml.xsd.types.XSDDateTime`
`xsd:decimal`	`java.math.BigDecimal`
`xsd:double`	`java.lang.Double`

XSD type in custom XSD	Maps to this Gosu type
xsd:duration	gw.xml.xsd.types.XSDDuration
xsd:float	java.lang.Float
xsd:gDay	gw.xml.xsd.types.XSDGDay
xsd:gMonth	gw.xml.xsd.types.XSDGMonth
xsd:gMonthDay	gw.xml.xsd.types.XSDGMonthDay
xsd:gYear	gw.xml.xsd.types.XSDGYear
xsd:gYearMonth	gw.xml.xsd.types.XSDGYearMonth
xsd:hexBinary	byte[]
xsd:int	java.lang.Integer
xsd:integer	java.math.BigInteger
xsd:language	java.lang.String
xsd:long	java.lang.Long
xsd:negativeInteger	java.math.BigInteger
xsd:nonNegativeInteger	java.math.BigInteger
xsd:nonPositiveInteger	java.math.BigInteger
xsd:normalizedString	java.lang.String
xsd:positiveInteger	java.math.BigInteger
xsd:short	java.lang.Short
xsd:string	java.lang.String
xsd:time	gw.xml.xsd.types.XSDTime
xsd:token	java.lang.String
xsd:unsignedByte	java.lang.Byte
xsd:unsignedInt	java.lang.Integer
xsd:unsignedLong	java.lang.Long
xsd:unsignedShort	java.lang.Short

In contrast, the following table lists the type conversions when using a Guidewire XML model to export an type from Gosu to XML. The left column is the Gosu type. The right column is the XSD type that Gosu natively defines for this type. If you want to create an additional (different) mapping for a Gosu type, create a new Guidewire XML model for that type. For more information about Guidewire XML models, see “The Guidewire XML (GX) Modeler” on page 262.

Gosu type	Guidewire XML Model maps to this XSD type
boolean	xsd:boolean
byte	xsd:byte
byte[]	xsd:base64Binary
char[]	xsd:string
double	xsd:double
float	xsd:float
gw.xml.xsd.types.XSDDate	xsd:date
gw.xml.xsd.types.XSDDateTime	xsd:dateTime
gw.xml.xsd.types.XSDDuration	xsd:duration
gw.xml.xsd.types.XSDGDay	xsd:gDay
gw.xml.xsd.types.XSDGMonth	xsd:gMonth
gw.xml.xsd.types.XSDGMonthDay	xsd:gMonthDay
gw.xml.xsd.types.XSDGYear	xsd:gYear
gw.xml.xsd.types.XSDGYearMonth	xsd:gYearMonth
gw.xml.xsd.types.XSDTime	xsd:time

Gosu type	Guidewire XML Model maps to this XSD type
int	xsd:int
java.lang.Boolean	xsd:boolean
java.lang.Byte	xsd:byte
java.lang.Double	xsd:double
java.lang.Float	xsd:float
java.lang.Integer	xsd:int
java.lang.Long	xsd:long
java.lang.Short	xsd:short
java.lang.String	xsd:string
java.math.BigDecimal	xsd:decimal
java.math.BigInteger	xsd:integer
java.net.URI	xsd:anyURI
java.net.URL	xsd:anyURI
java.util.Calendar	xsd:dateTime
java.util.Date	xsd:dateTime
javax.xml.namespace.QName	xsd:QName
long	xsd:long
short	xsd:short
gw.api.financials.CurrencyAmount	xsd:string
any Guidewire typekey	xsd:string

XSD Namespaces and QNames

The XML parser supports XSD qualified names (`QName` objects) with the `xsd:QName` XSD type. The XSD standard uses QNames to represent values within a specific namespace. A QName property on `XMLNode` objects sets or gets the namespace information. Gosu handles namespace conflicts automatically at serialization time.

If you want to create a `javax.xml.namespaces.QName` in Gosu, use code such as the following:

```
new QName( "http://xsd.mycompany.com", "weather" )
```

The first parameter is the URI. The second parameter is the *local* part of the qualified name.

If you want to additionally specify the prefix within the `QName`, use the following alternate method signature with an additional parameter for the prefix. For example, if you want to use the prefix `mine`, use the following:

```
new QName( "http://xsd.mycompany.com", "weather", "mine")
```

If you set an `xsd:QName` property, Gosu automatically declares the namespace at the appropriate level during serialization if needed. In the case of prefix conflicts, Gosu automatically generates unique namespace prefixes. If you let Gosu generate the unique namespace prefixes, these may change. Do not rely on these prefixes generating the same in future releases.

Autocreation of Intermediate Nodes

If a XSD-Based XML node refers to another XSD-Based XML node and you set a property on a subobject, Gosu automatically creates intermediate nodes if they do not yet exist. This allows you to assign to very long paths (dot-paths) without manually creating intermediate nodes.

XML Node IDs

Gosu supports node IDs and node ID references in XSDs and converts them into Gosu types. Node IDs are text values that uniquely identify an element. Some XSDs use node ID references to link to another element in the XML. In the XSD specification, node ID references (links to a node ID) do not reference the type of object, so

Gosu cannot automatically guess the type. However, Gosu provides an intuitive way to dereference a node ID reference using the `as` keyword.

Gosu converts the following XSD types as follows:

- A schema element or attribute of type `xsd:IDREF` becomes a property of type `IXMLNodeWithID`.
- Any element with an attribute of type `xsd:ID` automatically implements the interface `IXMLNodeWithID`.

At run time, you can populate the `IDREF` property with a link to an element with an ID (element with an attribute of type `xsd:ID`). To do this, simply assign an object that implements `IXMLNodeWithID` to an `IDREF` property. The IDREF property links to the property by its ID. There is no danger of infinite recursion.

If you do not set an ID explicitly, Gosu automatically assigns an ID. If a parsed XML document has automatically-assigned node IDs and you reserialize the XML, the result retains the original ID values unless there are ID conflicts within the document.

You can get the XML node that represent a node ID references directly from Gosu like any other node from Gosu. With this object, you can use the usual node actions `IXMLNode` actions documented earlier in this topic. However, typically you want a Gosu reference **to the linked-to object** so you can get its properties or subobjects. To do this, cast the ID reference node to the appropriate `XMLNode` type. In other words, cast the IDREF to the Gosu type representation of the linked-to object. The cast operation uses the ID reference value and the destination type to find and return the desired element with that ID. If the XML graph does not contain an element of the desired type with that node ID, Gosu throws an exception.

For example, suppose you want to import the following XSD, called `mytest.xsd`. Create a file by this name within your configuration module hierarchy in the package hierarchy `test`:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="root">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="category" maxOccurs="unbounded" minOccurs="0"/>
                <xsd:element name="item" maxOccurs="unbounded" minOccurs="0">
                    <xsd:complexType>
                        <xsd:attribute name="itemName" type="xsd:string"/>
                        <xsd:attribute name="categoryRef" type="xsd:IDREF"/>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="category">
        <xsd:complexType>
            <xsd:attribute name="categoryId" type="xsd:ID"/>
            <xsd:attribute name="categoryName" type="xsd:string"/>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

Notice the two lines that are bolded. The item element contains an `IDREF` called `categoryRef`. For the sake of this example, this will link to a `category` element within the XML. However, note that the definition of the `categoryRef` attribute does **not** specify the type of the linked-to object:

```
<xsd:attribute name="categoryRef" type="xsd:IDREF"/>
```

Next, create an XML file that uses that XSD. The following example has two category elements and one item element. The item links to one of the category objects using its `categoryRef` attribute (the IDREF). That IDREF corresponds to the `categoryID` property on a category. Save the following file to the path `/my_xml_files/mytest.xml`.

```
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="mytest.xsd">
    <category categoryId="FOO1" categoryName="category #1"/>
    <category categoryId="FOO2" categoryName="category #2"/>
    <item categoryRef="FOO1" itemName="item #1"/>
</root>
```

You can then create Gosu code that parses your XML against this XSD. For example, you can use the following code in the Gosu Tester:

```
uses java.io.File

// Load the XML:
var f = new File( "/my_xml_files/mytest.xml")

// Parse the XML
var xml = test.mytest.root.parse( f , true )

// get a property from the item node directly (does not use the IDREF)
print( xml.items[0].itemName)

// get the item node IDREF and cast to the category objects
var category = xml.items[0].categoryRef as gw.mytest.category

// you now have a Gosu reference to the element, so you can get properties from it:
print( category.categoryName )
```

Date Handling in XSDs

All of the XSD types related to dates (`xsd:dateTime` and related types) have Gosu counterparts in the package `gw.api.xml.xsd.types`. For example, the `XSDGDay` XSD type has the corresponding Gosu type `gw.xml.xsd.types.XSDGDay`.

All of these date types share the common superclass `AbstractXSDDateType`. You can construct all of them from a `java.lang.String` in the standard XML Schema date/time formats specified in the W3C XML Schema specification. Call the `toString` method on any of them to return the same format.

You can alternatively construct one of these from a `java.util.Calendar` object, passing a `boolean` specifying whether or not to include the `Calendar` object's time zone. The meaning of an `xsd:dateTime` without time zone information is application-specific.

You can call the `toCalendar` method on a date type to convert back to a calendar object. However, Gosu throws an `IllegalStateException` if you do not include time zone information. You can call the `date.toCalendar(tz : TimeZone)` method to specify a default time zone to use if a time zone does not exist in the `xsd:dateTime` object. If you use the `toCalendar` method, Gosu sets all `Calendar` fields to defaults except the fields that are significant to the `xsd:dateTime` type specification.

The Guidewire XML (GX) Modeler

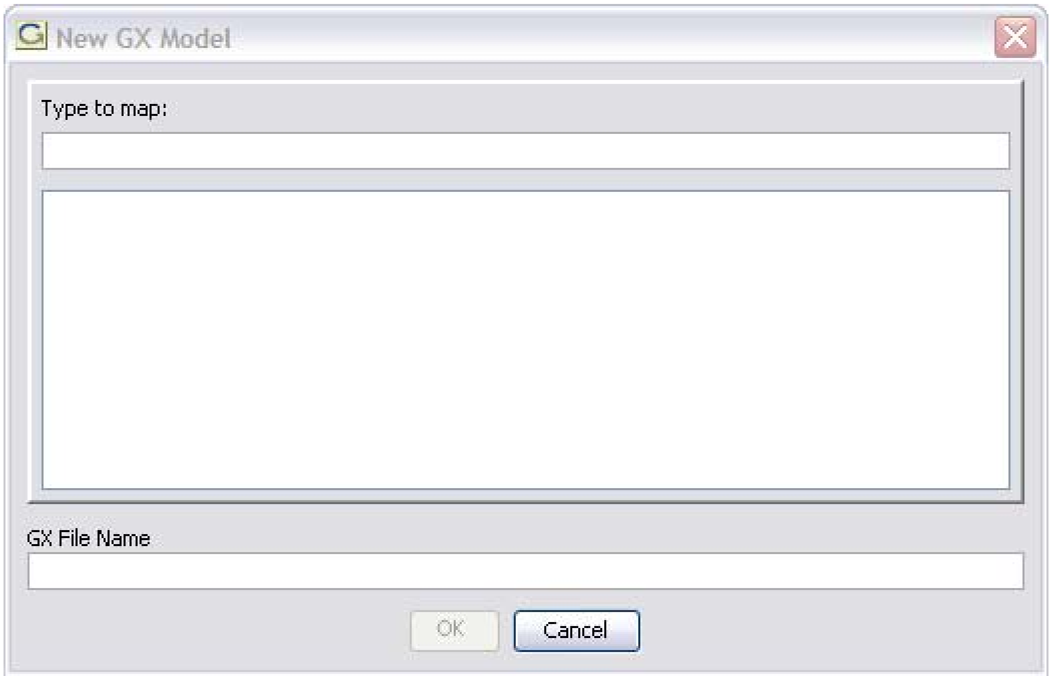
You can export business data entities, Gosu class data, and other types to a standard Guidewire XML format. You can select which properties to map in your XML model. By specifying what to map, ClaimCenter creates a XSD to describe XML that conforms to your XML model. At run time, you can export XML for this type and optionally choose to export only data model fields that changed. If you have more than one integration point that uses a type, you can create different XML models for each type.

To use your XML model in your messaging code, edit your Event Fired rules to generate a message payload using your XML model. Any generated XML using your XML model automatically conforms to the XSD and only includes the properties specified in your custom XSD.

The first step is to create a new XML model in Studio. In Studio, navigate in the resource tree to the location **configuration** → **Classes**. Within the classes tree, navigate to the package in which you want to store your XML model, just like you would for creating new Gosu classes. If you need to create a top-level package, right-click on the folder icon for **Classes**, and select **New Package**. Next, right-click on the desired package. From the contextual menu, choose **New** → **Guidewire XML Model**.

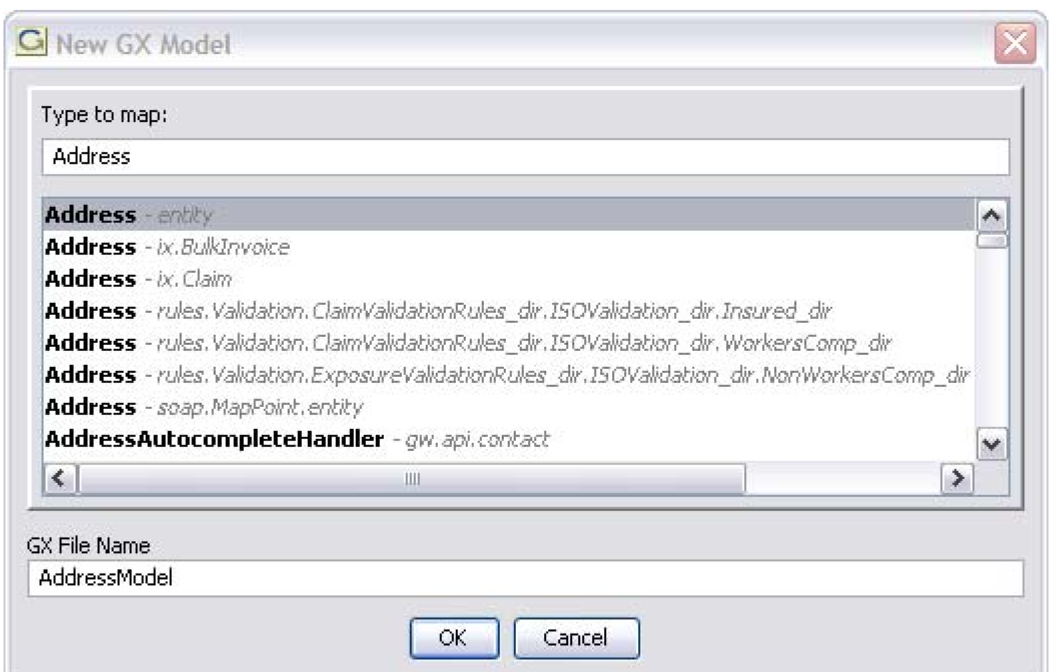
Sometimes in code or in other documentation, you might notice the Guidewire XML Model abbreviated in documentation as GX. GX is a shortened form of the name Guidewire XML. For example, GX models and the GX modeler tool.

Studio displays a dialog box that looks like the following:



Type the name of the type you want to export in XML. For example, to map and export an `Address` entity, type `Address`. The dialog box lists types that match the partial name you type.

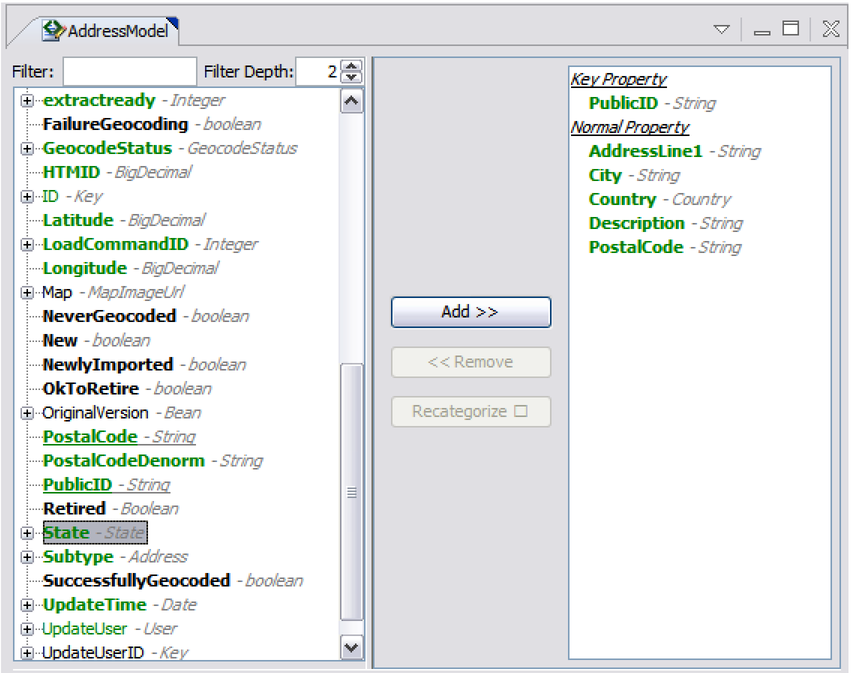
It is common for the list to include multiple types with identical names but different packages (namespaces) listed in gray. Be sure to choose the correct type in the list during this step.



By default, the file name for the XML model file is the type name followed by the word `Model`. For example, if the type is `Address`, the file name is `AddressModel`. Optionally, you can change the file name to something using the property at the bottom of the window.

Choose this name carefully. It defines the package hierarchy for your XML model. For example, if you created an XML model in the `mycompany.messaging` package and called the file `AddressModel`, the fully-qualified type for the `Address` entity in your type model is `mycompany.messaging.AddressModel.Address`.

Finally, click OK to open the XML model editor. It looks like the following:



Each property in the object graph appears differently based on the property type:

- **Mappable types.** A type is mappable if it is one of the following:
  - All simple property types such as `String`, `Date`, numbers, typecodes, and enumeration values are mappable. For a reference of how these property types map to XML, refer to “Gosu Type to XSD Type Conversion Reference” on page 258.
  - Any types you map with Guidewire XML Model editor become mapped types. If you attempt to map a property with a type that you have mapped with the Guidewire XML model editor, Studio displays a dialog box to select the desired model. If you created multiple XML models for that type, the dialog box contains multiple choices.

  The model editor displays these properties with bold text. Any field that is mappable and also a field in the data model, the model editor displays with green colored text.
- **Unmappable types.** Unmappable types are types that the Guidewire XML model editor cannot directly map:
  - Initially, complex types like custom Gosu classes and business entities such as `Claim` are unmappable. If you use the Guidewire XML model editor to create an XML model for that type, that type becomes mappable.
  - Even if a type is unmappable, a **property** on that type is mappable if it has a mappable type. For example, by default `Claim` entities are initially unmappable but any `String` property is mappable. Similarly, a property of a subobject is mappable if it has a mappable type, even if the object as a whole is unmappable.
  - If an unmapped type has no public properties with a mappable type nor any subobject has a mappable property, Studio hides properties of that type. For example, suppose a type called `Leaf` has no public properties or subobjects. Let us also suppose a type called `A` has a property called `MyLeaf` of type `Leaf`. On the type called A, Studio hides its `MyLeaf` property. There is no valid data that the Guidewire XML model edi-

tor can map for that property. The `Leaf` type as a whole is unmappable and none of its properties have mappable types, nor does it have subobjects with mappable types.

The model editor displays these properties with standard weight (non-bold) text. For unmappable field that are in the data model, the model editor displays them as green colored standard weight text.

You can make an unmappable type mappable by creating an XML model for it. For example, in a `Person` entity graph, a property that contains an entire `Address` entity is unmappable by default. If you create a model for the `Address` entity, you can then map the `Person` property that contains an entire `Address` entity.

Note: You must ensure that the type has at least one public property or one of its subobjects has a public property. Otherwise, you cannot create an XML model for that type.

However, you do not need to create an XML model for a type to map individual properties on it or subobjects of it. For example, suppose `Person.PrimaryAddress` has type `Address`. Even if you have not added an XML model for `Address`, you can map `Person.PrimaryAddress.City`, which is a `String` value.

Studio supports all variants of properties, including the following:

- Entity properties defined in the data model configuration files backed by database columns
- Entity virtual properties (properties not backed by database columns)
- Public properties on Gosu classes
- Public variables on Java classes
- Public properties implemented with Gosu enhancements (see "Enhancements" on page 209)

Note: Methods (functions) do not count as properties. Methods never appear in the property mapping hierarchy. If you want the return result of a method to be a mapped property, add a Gosu enhancement property (a `property get` function) to return the desired value.

For entities such as `Claim` and `Address`, some properties are backed by database columns in the Guidewire data model. However, whether a column is backed by a database column does **not** affect whether that property contains a mappable type. Simple properties like numbers and `String` values are automatically mappable. Foreign key references to other entities by default are not mappable by default. However, you could map properties within that subobject entity, or you can create an XML model for that subobject. As you export XML at run time, there can be behavior differences depending on whether a property is backed by an actual database column. For more information, see "Generating XML Using an XML Model" on page 267.

Initially, every property is unmapped in an XML model. In other words, initially an XSD contains no properties and any generated XML for that type with that model contains no properties. You can use the Guidewire XML model editor window to map as many properties as desired, including properties multiple levels down in the hierarchy. For every property with a mappable type, you can map the property in one of the following ways:

- **Normal property.** A normal mapped property can cause a type to export in XML. However, whether it actually exports or not is affected by other run time flags, discussed later.
- **Key property.** A key property always exports if that type exports due to changes to that object or a subobject. However, the presence of a key property only identifies the element. Defining a key property **never causes** an element or ancestor elements to generate in XML if the object would not otherwise generate in XML. The typical use of a key property is a unique identifier for an object. This is the reason for the name *key property*, similar to a *key field* in a database.
  - For Guidewire business entities, the `PublicID` property is the main key property that identifies the entity to an external system. Typically an external system needs the `PublicID` property, but only if there is a reason to export the object. For example, if important property changed on it or the object is new or added to an entity array. For an incremental export (only changed data), if an entity does not change and no subobject changes, Gosu does not export the object.
  - You can define a property as a key property even if it is not an actual unique identifier key for the object. For example, suppose a type has a debugging-related enhancement property on the `Claim` entity that calculates a value using a complicated formula using other properties. If it is useful to send that value to the external system, but **only** if you send the object, map this property as a key property.

If a type contains no `public` properties (including Gosu enhancements), a property of that type never appears in the left pane. If you want to make this type mappable in the Guidewire XML model editor, you must add at least one mappable public property to that type. If that type does not contain any properties yet, a simple way to do this is to add a Gosu enhancement on that type. Add a `property get` function to get your mappable data.

Your choice of what properties to map affects the XSD. However, whether you define a property as normal or key affects any run time XML output but does not affect the XSD. However, the actual XML that Gosu generates for an object depends on some configuration settings at run time. For example, these settings define whether to send the entire object graph or only the properties that changed. For details, see "Generating XML Using an XML Model" on page 267.

How to Map a Property

To map an unmapped property, first click the property to select it. You might need to navigate down through other properties. If so, click the "+" sign icon next to a property to open the hierarchy underneath it.

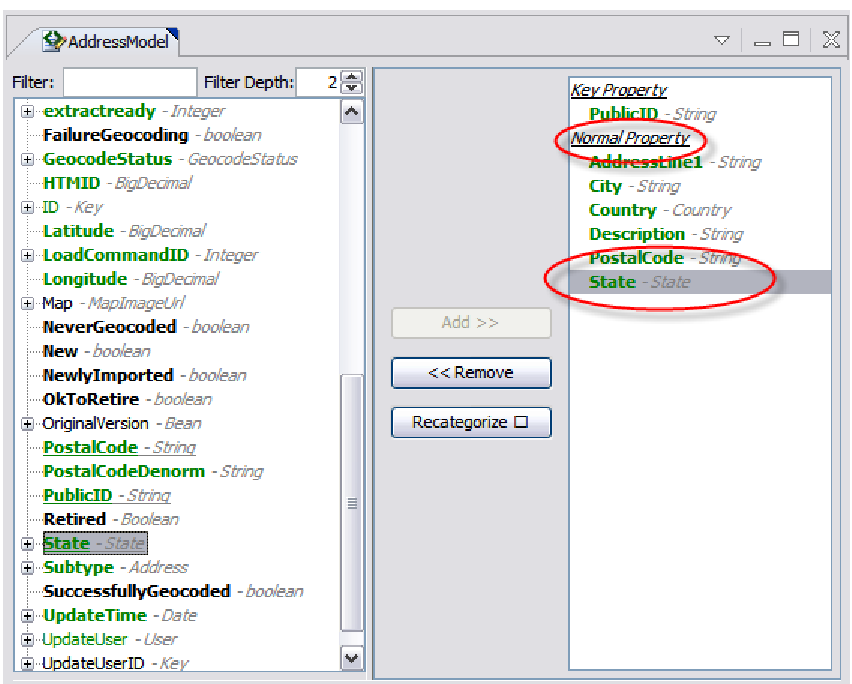
If you do not want to navigate through the entire hierarchy, you can instead use the search box to find the desired property by name. Studio updates the tree view to include properties with the name you typed if it is within the depth level from the root of the type. For example, suppose you have a type called `MyClass` and you need to access the property `MyClass.Property1.SubProperty.SubSubProperty`. Perform the following:

1. Notice that the desired property is three levels down in the entity path from the root class for the model. Thus, set the filter depth to 3 in the text box.
2. Type `SubSubProperty` into the text fields next to the word **Filter:**.
3. Studio updates the tree view to include properties with that name if it is within the depth level from the root of the type. There may be more than one property that fits that requirement.

Click the property in the list you want to add.

Next, click the **Add>>** button. Studio displays a popup that lets you choose one of the mapping types: Key Property or a Normal Property. Select one of these choices. Studio updates the window to show your property listed under the desired category in the right half of the window.

For example, the following window shows the `Address.State` property as a Normal Property:



Repeat this process for all properties you want in your XML model. You can preview the sample XML this model creates in the bottom pane of the window. The bottom pane has two tabs on the right side to toggle between the XML model (the **Sample XML** tab) and the XSD (the **Schema** tab).

After you add all properties for this XML model, you are ready to use this XML model in any Gosu code to convert an entity into an XML representation.

Automatic Publishing of the Generated XSD

When the ClaimCenter application is running, the application publishes the XSD. Other applications can import this XSD from ClaimCenter and validate any generated XML against this XSD.

ClaimCenter publishes the XSD at the following URL:

```
http://HOSTNAME:PORT/cc/service/xsd
```

The web server display a list of XSDs. Click on an XSD to view it. If your external system imports XSDs using an HTTP URL, save that URL to the XSD. If the external system needs the raw XSD on disk, copy the XSD contents and paste into another application as needed. You can also get the XSD from within Studio in the XML Model editor for your model.

Generating XML Using an XML Model

After you create an XML model for a type, you can easily generate XML for that type. You can access the XML model with classes that Gosu creates at the fully-qualified path using the package hierarchy in which you added your XML model.

For example, given the XML model file name *MODELNAME* and type name *TYPENAME*, access the type constructor using the following syntax

```
var xml = new YOURPACKAGE.MODELNAME.TYPENAME( object )
```

The example earlier in this section created a model for the `Address` entity with the default file name `AddressModel`. Let us assume you created this XML model in the package `mycompany.messaging`.

To map an instance of the type into XML, create an instance of the XML model class and pass the object to map as an argument to the constructor. For example:

```
var xml = new mycompany.messaging.AddressModel.Address( myAddress )
```

The variable `xml` contains the desired XML object.

Exporting XML Data

There are multiple ways to output the bytes from an XML node using other methods on the node, including the the `Bytes` property (the preferred method) and the `asUTFString` method.

> **IMPORTANT** For examples of XML export and issues related to Unicode and Guidewire messaging, see “Exporting XML Data” on page 246.

The examples in this section by default generate the entire XML graph. If you are using a Guidewire Standard XML model (see “The Guidewire XML (GX) Modeler” on page 262), you can choose additional export options:

- You can selectively export elements even for `null` properties using the `Verbose` property in `GXOptions`. For details, see “Customizing Guidewire Model XML Output (GXOptions)” on page 268.
- You can selectively export only entity properties that changed using the `Incremental` property in `GXOptions`. For details, see “Customizing Guidewire Model XML Output (GXOptions)” on page 268.

Customizing Guidewire Model XML Output (GXOptions)

Alternatively, you can use an alternate form of the constructor to add additional options that change XML export behavior. The alternative constructor takes an additional argument of type `GXOptions`, which stands for *Guidewire XML model options*. A `GXOptions` object has the properties described in the following table.

GXOption Property	Description	Behavior for properties backed by database columns	Behavior for properties not backed by database columns
`Incremental`	For properties backed by database columns, export properties only if they change. Default is `false`.	If set to `true`, exports the property only if it changes. To determine whether the property changes, ClaimCenter uses the current bundle to examine what entities were added, removed, or changed. **IMPORTANT:** There are important exceptions for this option's behavior, discussed later in this section If set to `false`, generates a property's XML even if it did not change.	No effect. The incremental feature does not apply to this type of property.
`Verbose`	Export an XML element for an property even if the value for that property is `null` The default for `Verbose` is `false`. This is the behavior if you use the one-argument constructor without the `GXOptions` object.	If the `Verbose` option is `true`, any `null` values export an element with the attribute `xsi:nil` set to `"true"`. This XML element never has any subelements in the XML. For example, the following XML represents the property `mytype` with a value `null`: `<mytype xsi:nil="true">` If the `Verbose` option is `true`, the special attribute is always on the highest-level object in the hierarchy that is `null`. Any subobjects or properties with implicitly `null` values are not in the output XML at all. For example, suppose the Gosu expression `myobject.A.B` evaluates to `null` because `myobject.A` is `null`. In this case, the `xsi:nil` attribute is on the `myobject.A` element and no element in the XML explicitly represents the B property. If the `Verbose` option is `false`, a `null` value does not export an element in XML.	Same behavior as for database columns. See previous table cell.

GXOption Property	Description	Behavior for properties backed by database columns	Behavior for properties not backed by database columns
`SupressExceptions`	Suppresses exceptions that occur in the process of getting values from objects. Exceptions can occur because some properties are actually backed either by Gosu enhancement `property get` functions or internal Java code (virtual properties). For a related topic, see "Checking for Exceptions in Property Export" on page 270. The default is `false`	Suppresses exceptions if true.	Suppresses exceptions if true.

The behavior is slightly different for any properties that satisfy both the following criteria:

- The property is a virtual property or enhancement property. In other words, the property is not defined in the data model configuration file and backed by a database column.
- Within that the object graph starting at that property there exists at least one object of a Guidewire entity type. This section refers to these entities as *entities inside virtual property graphs*.

In this case, Gosu cannot determine whether the entities inside virtual property graphs changed.

For example, in ClaimCenter, a claim's primary address is a virtual property on the `Claim` entity. Any subobjects of `Claim.PrimaryAddress` cannot be tested for new/deleted/changed entities.

For this special case, Gosu changes the behavior of entities inside virtual property graphs. Even if you set `Incremental` to true, the incremental mode is disabled for entities inside virtual property graphs. In other words, as Gosu recursively processes the type, if it finds a virtual property, incremental mode is turned off for that hierarchy of objects. All entities inside virtual property graphs ignore the `Incremental` option and thus generate XML even if unchanged.

Adding GXOptions

You can set `GXOptions` options using a compact syntax in Gosu for initializing a new `GXOptions` object. For details of data structure initialization syntax, see "Object Initializer Syntax" on page 71. The following example sets both `Incremental` and `Verbose` to `true`:

```
var xml = new mycompany.messaging.AddressModel.Address( myAddress, new GXOptions() {
  :Incremental = true , :Verbose=true
} )
```

Send a Message Only If Data Model Fields Changed

After you pass a type to your XML model's constructor, check the `ShouldSend` property on the result to determine. That returns true if and only if at least one mapped data model field changed in the local database bundle. If you only want to send your message if data model fields changed, use the `ShouldSend` property to determine whether data model fields changed.

For example:

```
if (MessageContext.EventName == "AddressChanged") {
  var xml = new mycompany.messaging.AddressModel.Address( MessageContext.Root as Address)
if (xml.ShouldSend) {
    var strContent = xml.asUTFString()
```

```
      var msg = MessageContext.createMessage(strContent)

      print("Message payload for debugging:")
      print(msg);
    }
  }
```

Note: When Gosu checks for mapped data model properties that changed, Gosu checks both normal or key properties. Of course, generally speaking, if your key property is supposed to uniquely identify the object, it never changes.

There is an important exception. If any ancestor of a changed property changed its value (including to `null`), it might not necessarily trigger `ShouldSend` to be `true`. If the original value of the non-datamodel field indirectly referenced data model fields, those fields do not count as properties that trigger `shouldSend` to be true. If the property that changed has a path from the root type that includes **non-datamodel fields**, then Gosu's algorithm for checking for changes does not see that this field changed. This is true also if the property changed to the value `null`. In other words, ClaimCenter can tell whether a property changed only if all its ancestors are actual data model fields (rather than enhancement properties or other virtual properties).

For example, suppose that due to some user action, some mapped field `A.B.C.D` property changed and also `A.B.C` became `null`. If the `B` property and the `C` property were both data model fields, then Gosu detects this bundle change and sets `ShouldSend` to true. However, if the `B.C` property is implemented as a Gosu enhancement or an internal Java method, then `ShouldSend` is `false.` This is because the bundle of entities do not contain the old value of `A.B.C`. Because Gosu cannot see whether the property changed, Gosu does not count it as a change that sets the property `ShouldSend` to the value `true`.

This special behavior with non-datamodel properties subgraphs is similar to how the `Incremental` property in `GXOptions` is effectively disabled within virtual property subgraphs. See related discussion about virtual property graphs earlier in this topic.

Note that if a property contains an entity array and its contents change, `ShouldSend` is `true`. In other words, if an entity is added or removed from the array, `ShouldSend` is true.

The behavior of `ShouldSend` does not change based on the value of the `Verbose` option or the `Incremental` option.

Checking for Exceptions in Property Export

When you use your model to generate XML from an instance of the type, exceptions can occur in some cases. Exceptions can occur because some properties are actually backed by Gosu enhancement `property get` functions or backed by internal Java code (virtual properties).

You can tell Gosu to suppress exceptions during XML generation using the `GXOptions` option called `SupressExceptions`.

After generating XML, you can check whether exceptions occurred using the `HasExceptions` property on the root XML object (the result of the constructor of the type using your model). If you want details of each exception, call the `eachException` method, which takes a Gosu block. Your block must take one argument, which is the exception. Gosu calls this block once for each exception.

Parsing XML Into an XML Model

You can use your model to convert XML data (the text) back into a Gosu typesafe graph of XML nodes based on `XMLNode`.

```
var myXMLData =
   "<Date xmlns='http://guidewire.com/gx/gxdemo.DateModel'>" +
   "<DayOfWeekName>Friday</DayOfWeekName>" +
   "</Date>"
var xml = gxdemo.DateModel.Date.parse( myXMLData )
```

Arrays of Entities in XML Output

For properties that represent arrays of entities in the Guidewire data model, there are special rules that govern the behavior of XML output.

Detecting Add and Remove

There is special formatting in the XML to represent added entities or removed entities from the array. This feature only applies if you use the incremental mode with the `Incremental` option.

> **Note:** For details of the `Incremental` option, see "Generating XML Using an XML Model" on page 267.

In the data model, for entity arrays, the actual foreign keys are on the child entities pointing to the parent entity. Gosu hides this implementation detail and makes the child entities appear as a read-only array on the parent entity.

If the `Incremental` option is `true`:

- If an element in an entity array is new in this database transaction, its element has the `action` attribute with the value `"ADD"`. Because this entity is new, this entity and its subobjects **fully export**. The output for this entity is complete, just as if incremental mode were set to `false`.
- If an element in an entity array was removed in this database transaction, its element has the `action` attribute with the value `"REMOVE"`.
- If an element in an entity array does not have the action attribute, then this element changed rather than was added or deleted.

Special Behavior for Entity Arrays Inside Virtual Property Graphs

The behavior is slightly different for any properties that satisfy both the following criteria:

- The property is a virtual property or enhancement property. In other words, the property is not defined in the data model configuration file and backed by a database column.
- Within that the object graph starting at that property there exists at least one object of a Guidewire entity type. This section refers to these entities as *entities inside virtual property graphs*.

In this case, Gosu cannot determine whether the entities inside virtual property graphs changed. Thus, for this special case, Gosu changes the behavior of entities inside virtual property graphs. All objects within the virtual property graph always export as if `Incremental` option were set to `false`.

There are other behaviors that are different for entities inside virtual property graphs. See "Generating XML Using an XML Model" on page 267 for details.

Complete Guidewire XML Model Example

The following is an example of creating an entirely custom Guidewire XML model for the `Date` type. After making a new Guidewire XML model, you can optionally map date fields with this instead of using the built-in support for standard XML/XSD date types.

To create a demonstration XML model for the date type

1. In Studio, right-click on **Classes**, select **New** → **Package**. Name the package `gxdemo`.
2. Right-click on the `gxdemo` package.
3. Select **New** → **Guidewire XML Model**
4. In the **Type to map** field, type `Date`. The list automatically selects `java.util.Date` as the most appropriate choice, unless you added any other similarly named classes already. The file name field at the bottom of the window automatically populates the name `DateModel`.
5. Click OK. The XML model editor appears.

6. Select the following properties from the list on the left part of the window: `DayOfWeekName`, `Hour`, and `Minute`. Use the control key to multiple-select all of them.

7. Click the **Add** button, and choose **Required** from the context menu that appears. The three properties appear in the list on the right.

8. Studio creates a file named `DateModel.gx` on disk. It has the following format:

```
<?xml version="1.0"?>
 <gx-model xmlns="http://guidewire.com/gx" type="java.util.Date">
 <includes>
 <include path="DayOfWeekName" fieldInclusionType="REQUIRED" type="java.lang.String"/>
 <include path="Hour" fieldInclusionType="REQUIRED" type="int"/>
 <include path="Minute" fieldInclusionType="REQUIRED" type="int"/>
 </includes>
 </gx-model>
```

This describes your new Guidewire XML model. At run time, Gosu turns this model into a real XML schema at runtime. The schema looks like this:

```
<?xml version="1.0"?>
 <xsd:schema
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:gw="http://guidewire.com/xsd"
             xmlns="http://guidewire.com/gx/gxdemo.DateModel"
             targetNamespace="http://guidewire.com/gx/gxdemo.DateModel"
             elementFormDefault="qualified">

     <xsd:element name="Date" type="Date"/>

     <xsd:complexType name="Date">
         <xsd:sequence>
             <xsd:element name="DayOfWeekName" minOccurs="0" type="xsd:string"
                gw:type="java.lang.String"/>
             <xsd:element name="Hour" minOccurs="0" type="xsd:int" gw:type="int"/>
             <xsd:element name="Minute" minOccurs="0" type="xsd:int" gw:type="int"/>
         </xsd:sequence>
     </xsd:complexType>
 </xsd:schema>
```

This schema describes an XML document that has only a single valid root element with name `Date` in the namespace `http://guidewire.com/gx/gxdemo.DateModel`. All elements in the XML exist in this same namespace. The `Date` element contains a sequence of three other elements:

- `DayOfWeekName`, of type `xsd:string`
- `Hour`, of type `xsd:int`
- `Minute`, of type `xsd:int`

9. Open the Gosu Tester and type the following program:

```
var xml = new gxdemo.DateModel.Date()
 xml.DayOfWeekName = "Monday"
 xml.Hour = 4
 xml.Minute = 30
 xml.print()
```

10. The program produces the following output:

```
<?xml version="1.0"?>
 <Date xmlns="http://guidewire.com/gx/gxdemo.DateModel">
   <DayOfWeekName>Monday</DayOfWeekName>
   <Hour>4</Hour>
   <Minute>30</Minute>
 </Date>
```

11. You can test populating the XML with an instance of the model's root type. In this example, an instance of the `java.util.Date` type. Type this program into the Gosu Tester:

```
var xml = new gxdemo.DateModel.Date( new java.util.Date() )
 xml.print()
```

When you run this program, it produces output similar to the following (varying by your actual local time):

```
<?xml version="1.0"?>
 <Date xmlns="http://guidewire.com/gx/gxdemo.DateModel">
   <DayOfWeekName>Wednesday</DayOfWeekName>
   <Hour>3</Hour>
   <Minute>28</Minute>
```

```
</Date>
```

12. You can test parsing XML data from an external system using this class. Type this program into the Gosu Tester in Studio:

```
var s = "<Date xmlns='http://guidewire.com/gx/gxdemo.DateModel'>" +
   "<DayOfWeekName>Friday</DayOfWeekName>" +
   "</Date>"
 var xml = gxdemo.DateModel.Date.parse( s )
 print( xml.DayOfWeekName )
```

13. This program generates the output:

```
Friday
```

14. You can use this model in another XML model. If you create another Guidewire XML model and try to map a `Date` property, Studio displays a dialog. In that dialog, you can choose which XML model to use, the default behavior for `Date` or your custom XML model.

Type Conversions from Gosu Types to XSD Types

For a full reference of Gosu types to XSD types used by the Guidewire XML model, see “Gosu Type to XSD Type Conversion Reference” on page 258.

chapter 19

Bundles and Transactions

Gosu provides APIs to change how (and at what times) changes to Guidewire entity save to your database. For many programming tasks in Gosu, such as typical rule set code, you do not need to know how database transactions work. However, for some situations you must understand database transactions. For example, you might need to do the following:

- add entities to the current database transaction
- move entities from one database transaction to another
- explicitly commit (save data) to the database
- purposely undo the current transaction by throwing Gosu exceptions.

WARNING Be extremely careful only to commit entity changes at appropriate times or you could cause serious data integrity errors. After an entity's changes commit to the database successfully, changes can no longer be rolled back. If an error happens in related code that executes after your code, all committed changes cannot be rolled back. However, in some cases all related code must commit changed data in a single transaction for proper application logic. For example, rule set code must never commit data explicitly because it happens automatically after all rule set code succeeds.

This topic includes:

Which Circumstances to Manipulate Database Transactions

Gosu provides APIs to change how (and at what times) changes to business data entities save to the database. For many programming tasks in Gosu, you do not need to know details of how database transactions work. For example, after rule set execution, related changes to the current objects are automatically committed (made final) in the database. If errors occur that prevent the entire action from safely completing, the entire transaction *rolls back*, which means to undo database changes that were tentatively made.

Situations that typically do not require special database transaction knowledge include:

- **Code that uses find queries and only reads properties.** If code uses `find()` queries and only **gets** values from entities retrieved from the database, you probably do not require special database transaction knowledge. See “Find Expressions”, on page 159 for related information.
- **Standard rule set code.** Rule set code that operates on a specific entity, such as validation rule sets or pre-setup rule sets. The application sets up a current database transaction and commits it if no errors occur. If you are changing only that entity and its subobjects, Gosu code does not typically require special actions related to database transactions.
- **Most plugin code.** Most plugin code in Gosu and Java that gets a specific entity as an argument to a plugin method has already been set up to use a specific database transaction. If you are changing only that entity and its subobjects, Gosu code does not typically require special actions related to database transactions.

  Note: There are exceptions in plugin development in Java where database transaction knowledge is necessary. For example, if you use Java threads, described in see “Running Code in an Entirely New Bundle” on page 282. Also if you create new entities, see the `EntityFactory` class discussion in “Java Entity Utility APIs” on page 121.

Situations that might require special database transaction knowledge include:

- **Code that uses find() and modifies found entities.** Code to **set** properties in entities retrieved from the database with a `find()` query must be aware of bundles. Entities are initially read-only if retrieved using a query but can be move to a writable memory location. See “Using Existing Bundles”, on page 277 for related database transaction information and “Find Expressions”, on page 159 for general `find` query information.
- **Web services written in Gosu that modify data.** If a web service API implementation modifies data, the API implementation must be aware of database transaction issues. Entities deserialized into Gosu as part of the incoming SOAP API call are automatically added to the current database transaction but are **not** automatically committed. To commit changes to the database, your code must explicitly commit the current collection of entities. If the SOAP API queries the database with `find()` statements or other APIs and wants to change entity data, you must add found entities to the current transaction. You explicitly must commit changes to the database. See “Using Existing Bundles”, on page 277 for related database transaction information.

  IMPORTANT Web service implementations automatically add deserialized SOAP entities to the current database transaction set up for the SOAP API but does **not** automatically commit entities passed to the web service.

  Note: In some cases, you might want to save some entities that were deserialized into the current transaction but not other entity changes that might occur. To handle this case, create a new database transaction and explicitly add entities to that group. For details, see “Running Code in an Entirely New Bundle” on page 282.
- **Asynchronous PCF or workflow code.** Background task code or self-contained code triggered from within PCF files or workflow definition files might perform operations that cannot rely on the current transaction set up by surrounding code. This current transaction is also called the *ambient transaction*. See “Running Code in an Entirely New Bundle” on page 282 for details.
- **Java code that spawns threads.** Bundles must not be shared across threads. See “Running Code in an Entirely New Bundle” on page 282 for details.

Committing data to the database can have tremendous effects on application logic and data integrity. You must be careful not to adversely affect other application logic that tracks the following:

- what time to commit changes
- what time to roll back (undo) tentative database changes.

WARNING Be extremely careful only to commit entity changes at appropriate times, or else serious data integrity errors occur. If an entity's changes commit to the database, changes cannot be rolled back even if errors later in related code. For example, all rule set code must never commit data explicitly. In many cases, such as rule execution, the database commit happens automatically after all related changes succeed.

Using Existing Bundles

To manage database transactions, Guidewire applications group business entities together in groups called *bundles*. Each bundle is a collection of one or more cached in-memory instances of entities to transmit and save to the database together. The set of entities include changed entities, new entities, and entities to delete from the database. Gosu represents bundles by the class `Bundle` in the package `gw.transaction`.

Guidewire refers to the process of sending the entities to the database as *committing the bundle* to the database. If any code attempts to commit the bundle and it completely succeeds, the database changes and finalize changes to all entities in the in a single database transaction. If the commit attempt fails in any way, the entire update rolls back (it is undone) and Gosu throws an exception.

Note: A bundle might never be committed. Errors in related code might prevent the current action from completing, and the bundle might be destroyed. If some code destroys the bundle with uncommitted changes, it is as if changes were forgotten, and entity data in the database was unchanged.

The most common thing to do with bundles is to add an entity to a bundle using the `Bundle.add(`*`entity`*`)` method. In most cases, Gosu might need to duplicate (clone) the in-memory instance of the entity during the transfer process. Unless the original entity was obtained from a `find()` query, you must assume the return result is a duplicate entity that belongs to the new bundle. Think of the return result of the `add` method as a copy that exists inside the new group of entities.

IMPORTANT In many cases, the return result of the `add` method is a cloned instance of what you passed into the method. If you want a reference to the entity or want to make further changes, you must keep a reference to the **return result** of the `add` method. Do not modify the original entity reference and do not keep a reference to the original entity.

However, if your original entity is in a read-only bundle, the return value of the `add` method is the same object as was passed in. The entity instance does not need to be cloned. If you got the entity from a `find()` query result, it is originally in a read-only bundle.

If at some later time the destination bundle commits and the entity was modified, the database saves the in-memory version in the destination of the entity to the database. If the entity was not yet in the database, the application adds the entity to the database as a new row in the table. You can also remove entities from the database, see "Removing Entities from the Database Entirely" on page 282.

In most cases, the destination bundle to use is the current bundle. Get the current bundle by calling the `getCurrent` static method on the `Transaction` class:

```
uses gw.transaction.Transaction

var bundle = Transaction.getCurrent()
```

With this bundle reference, you can call other methods on it like `add`.

The following example gets the current bundle, adds an entity to the bundle, and then changes a property:

```
uses gw.transaction.Transaction

// get the ambient (current) bundle from the current database Transaction
var bundle = Transaction.getCurrent()

// move (and possibly copy) the entity, being sure to save the result of the add() method
copyOfEntity = bundle.add(anEntityIWantToChange)

//if you modify properties, modify the new copy of the entity:
copyOfEntity.Description = "New Description Value"
```

If you want to change data you retrieved from a `find()` query, your code might look like the following:

```
// get the current (ambient) bundle of the runing code
var bundle = gw.transaction.Transaction.getCurrent()

var query = find( p in Claim.Policy where p.PolicyNumber == "54-123456" )

for( claim in query ) {
  bundle.add(claim)

  // make changes to one or more properties directly on the entity or its subobjects
  claim.MyProperty1 = true
}
```

Alternatively, you can get a bundle from an existing entity from its *`entity`*`.Bundle` property. You can call the `add` method on that bundle and modify the result if the bundle is not read-only (it is read only if it was a `find` query result). If you use this approach, **all** entities in the bundle commit to the database, not just the entity from which you got the bundle reference.

The following is an example of this approach:

```
// Get the bundle from an existing entity with code like:
var bundle = myClaim.bundle

// move (and possibly copy) the entity, being sure to save the result of the add() method
copyOfEntity = bundle.add(anEntityIWantToChange)

//if you modify properties, modify the new copy of the entity:
copyOfEntity.Description = "New Description Value"
```

Gosu requires that one of two things to be true at the time you attempt to add an entity to a bundle:

- The original entity is in a read-only bundle.
- The original entity is in a writable bundle but is not yet edited

If you try to add an entity in another bundle and the entity was modified, Gosu throws the exception `IllegalBundleTransferException`. Be careful not to let the entity be modified in more than one bundle. If it is already modified, Gosu throws an exception if you try to copy it to the new location, as mentioned earlier.

However, even if the entity was unmodified at the time you copy it to the new bundle and the entity exists in another writable bundle, there might be problems later. If both entities load from the same database record, only one bundle can try to commit the change to that entity. If an entity is modified in more than one bundle and both edited and committed, the second one fails with a concurrent data modification exception. That second commit attempt that fails might be in code other than yours, such as user interface code.

WARNING If an entity exists in a modified form in more than one bundle and both bundles commit, the second one fails with a concurrent data modification exception. Gosu attempts to avoid this problem by preventing adding an entity to a new bundle if it is already modified in memory, by throwing an exception during the `add` method. In some cases, user interface code internally refreshes an entity, which discards cached versions and gets latest versions. However, there is no supported API to refresh entities from Gosu.

Getting an Entity from its Public ID or Internal ID (Key)

In some cases you might not have a reference to the entity itself although you might have one of the following:

- The entity's public ID, which is its `publicID` property
- The entity's key, which is its unique internal `ID` property represented by the Gosu type `Key`. This internal ID is also the native ID used for foreign keys, such as the result of the `getOriginalValue` method to get a property that links to another entity.

In these common cases, get references to the entities to using the alternate methods `loadByPublicID` and `loadByKey`. These are especially helpful if you write web services implementations in Gosu. For web services that you write in Gosu, refer to entities by public ID if possible rather than attempting to pass a large entity and potentially large graphs of subobjects.

To load an existing entity by its public ID using `loadPublicID` with the entity type and the public ID. You will need to cast it to the correct (more specific) subtype using the `as` keyword:

```
uses gw.transaction.Transaction

var a = Transaction.getCurrent().loadByPublicID(Address, "ABC:1234") as Address
```

To load an existing entity by its `Key` with the entity type and an internal ID:

```
uses gw.transaction.Transaction

var a = Transaction.getCurrent().loadByKey(Address, "1234" as Key) as Address
```

The destination bundle must be writable (non-read-only) if you want to modify the data and save changes to the database. Edit the new entity as desired, and commit the bundle afterward if appropriate.

WARNING Be extremely careful only to commit entity changes at appropriate times, or else serious data integrity errors occur. If an entity's changes commit to the database, changes can no longer be rolled back if an error happens in related code that must undo related database changes. For example, all rule set code must never commit data explicitly since committing happens automatically if all related changes including rule set code succeeds.

Creating New Entities in Specific Bundles

If you pass arguments to the `new` operator, Gosu calls a specific constructor (a post-creation function) defined for that Java type. If there are multiple constructors, Gosu chooses the right one based on the arguments you pass it (the number of arguments and their types). For Guidewire business entities such as `Claim` and `User`, typical code passes no arguments to create the entity. Passing no arguments tells Gosu to create the entity in the current database transaction (the current *bundle*), which is usually the best approach.

In special cases, pass a single argument with the `new` operator to override the bundle in which to create the new entity. Pass an existing entity to creates the new entity in the **same** bundle as the entity you specify. Pass a bundle to explicitly use for the new entity. The following table compares different arguments to the `new` operator.

Expression	Result
`new Note()`	Constructs a new `Note` entity in the current bundle. Changes related to the current code submit to the database at the same time as Gosu adds the new note to the database.
`new Note(myClaim)`	Constructs a new `Note` entity given a `Claim` as the bundle to use. Changes to the `Claim` are submitted to the database at the same time as Gosu adds the new note to the database.
`new Note(myClaim.bundle)`	Constructs a new `Note` entity given a `Claim` as the bundle to use. Changes to the `Claim` submit to the database at the same time as Gosu adds the new note to the database.

For more information about the new operator, see "New Object Expressions" on page 70.

Committing a Bundle

Committing a bundle means to send all changes for entities in this bundle to the database and save their new values. Use the method *bundle*.commit() to attempt to commit the entire bundle. If it fails, it throws an exception and rolls back any changes. Either the entire transaction commits or the entire process fails and the database is unchanged.

The Transaction class has a static method getCurrent() that you can call to get the current bundle. For typical use, the current bundle is typically the bundle you want to add entities to or to commit. The Transaction class also contains other features, discussed in later sections.

Get the *current bundle*, which represents the current database transaction, by calling the getCurrent static method on the Transaction class:

```
uses gw.transaction.Transaction

var bundle = Transaction.getCurrent()
```

The following is a simple example web service that takes an entity and commits the entity's bundle:

```
uses gw.transaction.Transaction

@WebService
class AddressExampleAPI {
  public function insertAddress(newAddress : Address) : Address {

   // commit all entities in the current bundle
   gw.transaction.Transaction.getCurrent().commit()

   // note: for SOAP API implementations, bundle includes incoming enties
   // deserialized with the request.

   return newAddress
 }
}
```

If you have an entity reference, get its bundle from accessing the *entity*.bundle property. However, remember that committing a bundle commits **all** entities in the bundle, not just the one from which you got the bundle reference. For example:

```
var bundle = myClaim.bundle.commit() // commits *all* entities in the bundle of this object
```

WARNING Be extremely careful only to commit entity changes at appropriate times, or else serious data integrity errors occur. If an entity's changes are commit to the database, changes can no longer be rolled back in related code that must undo all related database changes. For example, all rule set code must never commit data explicitly since committing happens automatically if all related changes including rule set code succeeds.

For additional web services examples, see "Bundles and Web Services" on page 280.

Bundles and Web Services

Guidewire provides APIs related to database transactions. You may need to use these APIs if you design SOAP APIs that take parameters that must commit to the database. You may need to use these APIs if your web service looks up entities in the database and makes entity changes. These APIs are documented fully in "Bundles and Transactions" on page 275 in the *Gosu Reference Guide*. This section shows a simple example. See that topic for additional details and code samples.

A *bundle* is a group of Guidewire entities, grouped together so they save to the database together. You can get the current transaction by calling gw.transaction.Transaction.getCurrent().

The most important things to know about changing entity data in your web service as follows:

- When a web service client calls your web service, the system sets up a bundle automatically.

- Get the current bundle by calling `gw.transaction.Transaction.getCurrent()`. The current bundle is the bundle setup for your code. You can add entities to this writable bundle. For example, add read-only entities to this bundle.
- If you have a reference to an entity, get its bundle as the *`entity`*`.bundle` property. However, generally speaking use `gw.transaction.Transaction.getCurrent()`.
- Any entities serialized into the web service as arguments to your web service are already in the current bundle. You do not need to add them to the current transaction.
- No data is automatically commits to the database after your web service completes. If you want the existing transaction (including serialized data) to commit to the database you must do so manually by calling the `commit` method on the bundle.
- If you query the database with `find()` statements, entities you find are initially read-only. You must add them to the current writable transaction to modify the entities. Add them by calling the following code:

```
writableEntity = gw.transaction.Transaction.getCurrent().add(foundEntity)
```

- You can create an entirely new bundle to encapsulate a task in a Gosu block. A block is an in-line Gosu function contained in other Gosu code. See “Running Code in an Entirely New Bundle” on page 282.

  In your block, create new entities with the bundle argument as an argument to the entity constructor:

  ```
  writableEntity = new Address(bundle)
  ```

  In your block, add existing entities to the new bundle with the `bundle.add(`*`entity`*`)` method:

  ```
  writableEntity = bundle.add(foundEntity)
  ```

- There may be cases where web service method parameters include entities you do not want to commit but you make other entity changes that must commit. If so, create an entirely new bundle and explicitly add entities to that group, as discussed in “Running Code in an Entirely New Bundle” on page 282. Simply let the application commit the new bundle created in `runWithNewBundle`. If you use `runWithNewBundle`, do **not** commit the default (current) bundle for the web service.

The following simple example web service takes an entity:

```
uses gw.transaction.Transaction

@WebService
class AddressExampleAPI {
  public function insertAddress(newAddress : Address) : Address {

   // commit all entities in the current bundle
   gw.transaction.Transaction.getCurrent().commit()

  // note: for SOAP API implementations, bundle includes incoming entities
  //       deserialized with the request.

   return newAddress
 }
}
```

WARNING Be extremely careful only to commit entity changes at appropriate times or serious data integrity errors occur. If an entity’s changes commit to the database, changes can no longer roll back if errors happen in related code. Typically, errors must undo all related database changes. For example, Gosu code in rule sets must never commit data explicitly since the application automatically commits the bundle automatically only after all rule sets and validation successfully runs.

The following example web service gets a user and adds roles to it. Finally, it commits the result:

```
/**
* Adds roles to a User.
* If a role already belongs to the user, it is ignored.
*
* @param userID The ID of the user
* @param roleIDs The public IDs of roles to be added.
*/
  @Throws(DataConversionException, "if the userID or roleID does not exist")
  @Throws(SOAPException, "")
  public function addRolesToUser(userPublicID : String, roleIDs : String[]) : String {
```

```
    var user = User( userPublicID )
    if (user == null){
      throw new DataConversionException("No User exists with PublicID: " + userPublicID);
    }

    // add user to current bundle -- the current database transaction
    // this is always strictly required if you got the entity from a find() query!
    // we get the return value of add() and it has a DIFFERENT value than before.
    // It is a copy in the new bundle!
    user = Transaction.getCurrent().add( user );

    if (roleIDs == null || roleIDs.length == 0){
      throw new RequiredFieldException("Roles");
    }

    // Add in all the roles for the user
    for (var roleId in roleIDs) {
      var role = loadRoleByIdOrThrow(roleId);

      // pass an argument when creating a new entity with "new" to use a specific bundle
      var userRole = new UserRole();

      userRole.Role = role
      user.addToRoles(userRole);
    }

    // commit our bundle
    Transaction.getCurrent().commit();
    return userPublicID;
  }
```

Removing Entities from the Database Entirely

Instead of adding or changing entities by adding them to the bundle, mark an entity for removal from the database using the bundle `remove(...)` method. If the bundle commits, Gosu removes the entity from the database.

For example:

```
uses gw.transaction.Transaction

var a = Transaction.getCurrent().loadByPublicID(Address, "ABC:1234") as Address

Transaction.getCurrent().remove(a) // removes from database, not just from bundle
```

Be careful to understand that this does **not** simply remove entities from the bundle so that changes to the entity are forgotten. There is no API to evict an entity from the bundle. However, you can remove an entity from the database and the bundle using the `remove(...)` method.

There is no API to *move* an entity to another bundle, but you can use the bundle `add(...)` method to add an entity to another bundle, which in most cases duplicates the entity into the destination bundle.

WARNING The `remove` method is dangerous so use it with great care. Despite the name that seems to mirror the `add` method, this method does **not** simply remove the entity from the bundle's list of entities (no such API exists). The bundle `remove` method marks entities to remove the record from the database if this bundle is committed

Running Code in an Entirely New Bundle

Typical Gosu code interacts with database transactions by adding modifying entities in the current bundle or adding new entities to the bundle. The bundle and ClaimCenter automatically manage the low level parts of database transactions. Only in rare cases your code must commit a bundle explicitly. Refer to the previous section for more information about bundles and how to use them.

In very rare cases, you can run Gosu code in an entirely **new** database transaction. In other words, you sometimes must create a different transaction from the current transaction.

Only create your own transaction in the following cases:

- Create a new bundle if your code modifies entities to commit independent of the **success** of the surrounding code. For example, some workflow asychronous actions or PCF user interface code might need to create a new bundle.
- Create a new bundle if your code represents a separate asynchronous actions from surrounding code
- Create a new bundle if your Gosu calls Java code that spawns a new thread within the server, such as a timer or scheduled event. It is important that no two threads share a bundle.

To run code within a separate transaction, you must create a *Gosu block*, which is a special type of Gosu function that you define in-line within another function. You can pass a block to other functions to invoke as appropriate. For detailed information about blocks, see "Gosu Blocks", on page 213.

You can create a new bundle with a static method `runWithNewBundle` on the `Transaction` class. You pass the `runWithNewBundle` method a Gosu block that takes one argument of type `Bundle`, which is an entirely new bundle created just for your code within the block.

The syntax is:

```
uses gw.transaction.Transaction

...

Transaction.runWithNewBundle( \ bundle -> YOUR_BLOCK_CODE_HERE )
```

Your Gosu code within the block can add entities to the bundle as appropriate using the `bundle.add(entity)` method.

Gosu runs this code immediately (synchronously) and if the block succeeds with no exceptions, `runWithNewBundle` automatically commits the new bundle after your code completes. For the most concise and easy-to-understand code, Guidewire encourages you to use this automatic commit behavior rather than committing the bundle explicitly. If your code detects error conditions, it can roll back the database transaction associated with this bundle simply by throwing an exception from the Gosu block.

The following example demonstrates how to create a new bundle and run code that creates a new entity in that bundle, and modifies some fields. Note that the current transaction is an argument to the block, but it is redundant since Gosu sets the current (ambient) bundle inside the block to this new block automatically.

```
gw.transaction.Transaction.runWithNewBundle( \b -> {
  var me = new MyEntity()

  me.FirstName = firstName
  me.LastName = lastName
  me.PrimaryAddress = address
  me.EmailAddress1 = email
  me.WorkPhone = workPhone
  me.PrimaryPhone = primaryPhoneType
  me.TaxID = taxId
  me.FaxPhone = faxPhone
} )
```

The following example method from a web service implementation demonstrates how to create a new bundle and call an auxiliary function that takes a `MyClass` entity and a bundle:

```
...

public function myaction(f : MyClass) : void {
  Transaction.runWithNewBundle( \ bundle -> MyClass.doSomething(f, bundle))
}

...
```

In that example, the block was self-contained and did not need to transfer any information to the code within the `myaction` method.

In other cases, you might need to get information to your block or return information back from your block. You can do this using the *variable capturing* features of blocks. This feature shares variables in the current scope with the original calling context of the block and also code within the block. For more information about this feature, see “Variable Scope and Capturing Variables In Blocks” on page 216.

The following example demonstrates variable capturing by creating a local variable in the outer scope (`synchronizeAgency`) but then setting its value from within the block. Consequently, the outer scope can return that value as a return value from the method.

```
public function synchronizeMyObject(f : MyClass) : boolean {
    var res : boolean

    // the local variable "res" and the parameter "f" is captured by the block.
    // Both are effectively shared between the current function and the block
    Transaction.runWithNewBundle( \ bundle -> { res = MyClass.doSomething(f, bundle) })

    // return the value that was set in the block using the shared (captured) variable
    return res
}
```

Similarly, you can use variable capturing to provide additional information to your block. In the example earlier, notice that the variable `f` is a parameter to the outer function. It is captured by the block and can be read (or changed) by the block.

WARNING Be extremely careful only to commit entity changes at appropriate times, or else serious data integrity errors occur. If an entity’s changes commit to the database, changes can no longer be rolled back if errors occur that must undo related database changes. For example, all rule set code must never commit data explicitly since committing happens automatically if all related changes including rule set code succeeds.

Create Bundle For a Specific ClaimCenter User

The `gw.transaction.Transaction` class has an alternate version of the `runWithNewBundle` method to create a bundle with a specific user associated with it. This is intended for use in contexts in which there is no built-in user context, such in the startable plugins feature. The method signature is:

```
gw.transaction.Transaction.runWithNewBundle(\ bundle -> YOUR_BLOCK_BODY, user)
```

For that second method argument, you can pass either a `User` entity or a `String` that is the user name.

Warning about Transaction Class Confusion

There is more than one `Transaction` type in Gosu for ClaimCenter. Do not confuse the class `gw.transaction.Transaction` with the `Transaction` entities or related typelists.

If you use these APIs, you might want to use a Gosu `uses` statement such as the following:

```
uses gw.transaction.Transaction
```

This topic is about the class in the `gw.transaction` package, in other words: `gw.transaction.Transaction`.

Exception Handling in Transaction Blocks

Generally speaking, if your code throws an exception, any associated current database transaction rolls back. Any data associated with the current action does not commit to the database. Throw an exception from your code to get this behavior where appropriate. If you catch the exception, remember to rethrow the exception to get the roll back behavior.

This is true in cases where some code other than yours automatically handles committing the bundle. It is also true using the `runWithNewBundle` method. Throwing exceptions in the block prevents automatic bundle commits that happen after a most entity-related code completes successfully.

Determining What Entity Data Changed in a Bundle

If entity data changes in a bundle, it is sometimes important to identify which data changed. In other words, you might need to compare the most recent entity data from the database with the latest version of an entity in a locally-modified bundle. In the most common case, rule sets written in Studio contain rules and sometimes the rule conditions and rule actions must detect which, if any, properties changed in an entity. For example, if a certain entity property changed, you might want to recalculate other related properties, or perhaps log changes or send messages to external systems.

To detect such changes, you can call the entity's `isFieldChanged` method to find out if a certain property in that entity has changed. If it has changed, call the entity's `getOriginalValue` method to get the original value loaded from the database. You must case the original value to the expected type with the syntax "`as` *`TYPENAME`*". Depending on the type of value stored in that property, the `isFieldChanged` method behaves slightly differently, as described further in the following table:

Property type	Meaning of Original Value	Behavior of isFieldChanged(...)
Scalar value	The original simple value, such as a `String` value like `"John Smith"`, a number like 234, or any other non-entity and non-array type.	Returns `true` if and only if the actual property value changed to a different actual value. If the value changed to a different value and then back to the original value, `isFieldChanged` would return `false`.

Entity	Guidewire applications represent links to entity subobjects as a foreign key called the *internal ID*. This foreign key is the *`entity`*`.Id` property on the destination entity. Note that the `Id` property is different than the `PublicID` property.
	
	If you call `getOriginalValue` with a foreign key field, you get an internal ID. To get a reference in Gosu to the entity with the original ID, pass that ID as an argument to the entity type itself. That is a special syntax for looking up an entity from its internal ID. For example, suppose an entity in variable `e` has an `Address1` field that contains an `Address` entity. The following code gets the original value and a reference to it.
	
	`var f = e.getOriginalValue("Address1")`
	`var origAddress = Address(f)`
	
	Note that you do **not** use the `new` operator using this special entity lookup syntax (*object literals*) shown on the previous line.
		Returns `true` if and only if the foreign key `Id` value for the subobject is the same in the original entity. The `PublicID` is not examined by the `isFieldChanged` method.

Owned array	The original entity array	Returns `true` if an entity in the array changed (if one or more properties changed in an array) **or** if any elements were added or removed to the array. Because entity array order is not meaningful (do not rely on it), the order is not checked by `isFieldChanged`. Effectively, in this case `isFieldChanged` returns the same value as the array method `isArrayElementAddedOrRemoved`, discussed in "Bundle Change Methods Specific to Arrays" on page 287

Property type	Meaning of Original Value	Behavior of isFieldChanged(...)
Non-owned array	The original non-owned entity array	Returns `true` if and only if any elements were added or removed to the array. The set of elements is determined by the foreign key in the *`entity`*`.Id` properties of each array element). Because entity array order is not meaningful (do not rely on it), the order is not checked by `isFieldChanged`. Effectively, in this case `isFieldChanged` returns the same value as the array method `isArrayElementAddedOrRemoved`, discussed in "Bundle Change Methods Specific to Arrays" on page 287.

For a scalar value example, the `Address` entity contains a `String` value property called `City`. If the `address.City` value is different from the original entity, `address.isFieldChanged("City")` returns `true` and `address.getOriginalValue("City")` returns the original value for this property of type `String`.

For an entity example, the ClaimCenter application includes a `Claim` entity with a `LossLocation` property containing an `Address` entity.

- If that property points to an entirely different entity than the original entity loaded from the database (if it has the same `entity.Id` foreign key value), it is considered changed. Thus the expression `claim.isFieldChanged("LossLocation")` returns `true` and `claim.isFieldChanged("LossLocation")` returns the original entity and all its properties the original property values on the entity. For example:

```
if (claim.isFieldChanged("LossLocation")) {
  // get original entity
  var originalAddress = claim.getOriginalValue("LossLocation") as Address

  // get original property on original entity
  var origAddLine1 = originalAddress.AddressLine1
}
```

- If the `claim.isFieldChanged("LossLocation")` returns `false`, then the entity foreign key has not changed but that does **not** mean necessarily that data in its subobject were unchanged. If you want to check properties on subobjects, you must test specific properties on the subobject using the `isFieldChanged` method. For example:

```
if (not claim.isFieldChanged("LossLocation")) {
   if (claim.LossLocation.isFieldChanged("City")) {
     var origAddLine1 = claim.LossLocation.getOriginalValue("AddressLine1")
  }
}
```

Remember to call the `isFieldChanged` or `getOriginalValue` method on the correct entity, and note the differences in `isFieldChanged` for different value types.

IMPORTANT It is extremely important that you call the `isFieldChanged` or `getOriginalValue` methods on the correct entities and in the correct order in your code. In particular, remember that calling an entity's `isFieldChanged` method checks only that property. This method behaves differently for various types of values, and for entity values never detect changes in subobjects.

Detecting If Any Element Changes Occurred

You can use the read-only entity *`entity`*`.Changed` to check if the entity changed at all. It returns a Boolean value of `true` if the entity has one or more properties that changed.

```
if (myAddress.Changed) {
  // your code here...
}
```

Getting All Changed Properties From an Entity

You can get the entity's `ChangedFields` property to get a set of changed properties on this entity. This value is of type `Set` in the package `java.util`. For example:

For example, the following example uses the Gosu blocks feature to iterate across the list:

```
if (myAddress.Changed) {
  myAddress.ChangedFields.each( \ e -> print("address has changed property: " + e))
}
```

For more information about blocks and the `each` method, see "What Are Blocks?", on page 213 and "Collections", on page 231.

Bundle Change Methods Specific to Arrays

In addition to the support for `isFieldChanged` and `getOriginalValue` for entity array properties, call additional methods to test for changes to array contents in the bundle version of the entity. These methods compare the persisted entity to the cached version from the database.

> **IMPORTANT** The following bundle-related methods work only with entity array properties that are directly defined on an entity in the data model configuration files. They do not work any other types of properties or methods. For example, they do not work with Gosu enhancement properties, enhancement arrays, or other virtual properties (perhaps implemented in internal Java code). Refer to the Data Dictionary for details of whether a property is a native database-backed property or a virtual property.

The following table lists the available methods:

Method	Argument	Returns
`getAddedArrayElements`	`arrayFieldName`	Returns a list of array elements that were added, or an empty array if there were none. This works only with entity array properties that are directly defined on an entity in the data model configuration files. This method does not support Gosu enhancement properties, enhancement arrays, or other virtual properties.
`getChangedArrayElements`	`arrayFieldName`	Returns a list of array elements that were changed, or an empty array if there were none. This works only with entity array properties that are directly defined on an entity in the data model configuration files. This method does not support Gosu enhancement properties, enhancement arrays, or other virtual properties.
`getRemovedArrayElements`	`arrayFieldName`	Returns a list of array elements that removed (marked for deletion if the bundle commits), or an empty array if there none removed. This works only with entity array properties that are directly defined on an entity in the data model configuration files. This method does not support Gosu enhancement properties, enhancement arrays, or other virtual properties.
`isArrayElementChanged`	`arrayFieldName`	Returns `true` if and only if an element in an array changes, in other words if one or more properties change on the entity. If no existing element changes but elements add or remove, this returns `false`. This works only with entity array properties that are directly defined on an entity in the data model configuration files. This method does not support Gosu enhancement properties, enhancement arrays, or other virtual properties.
`isArrayElementAddedOrRemoved`	`arrayFieldName`	Returns true if and only if any array elements were added or removed. This works only with entity array properties that are directly defined on an entity in the data model configuration files. This method does not support Gosu enhancement properties, enhancement arrays, or other virtual properties.

Getting Add, Changed, or Deleted Entities In a Bundle

A bundle's list of entities might include some combination of the following types of entities:

- **New entities.** Gosu adds new entities to the database if this bundle commits. To get a list of these in a bundle, use the code `bundle.InsertedBeans`. It returns an iterator that iterates across these keyable entities.
- **Changed entities.** An entity may have one or more properties modified. To get a list of these in a bundle, use the code `bundle.UpdatedBeans`. It returns an iterator that can iterates across these keyable entities.
- **Deleted entities.** Some entities may be marked for deletion if this bundle commits. If the entity is retireable, the row remains in the database rather than deleted but the entity is marked as *retired*. To get a list of these in a bundle, use the code `bundle.RemovedBeans`. It returns an iterator that can iterates across these keyable entities.
- **Unmodified entities.** An entity might be added to a writable bundle but not yet modified. There is no API to get the list of unmodified entities in the bundle.

Bundle Commit and Query Implementation Details

How the Application Caches Entity Data and Prevents Problems

The application caches entity data for faster access. All APIs discussed in this topic that refer to *original* entity data only check against the database as of the time the bundle loaded that entity. The time of loading might not have been immediately before your code checks the original entity data. In many cases, the application may have loaded the entity from the database long before the current data access.

Although it is possible that the database data has changed since then, some safeguards prevent concurrent data access in most cases. For typical entity access, the server prevents entity commits if it changed in the database between the time the entity loaded in the bundle and the time the bundle commits.

Versioning on Entities

The mechanism for protection is an version property on *versionable entities*, which is a category that describes almost all entities in the system. If you load a versionable entity into a bundle, the application loads the version number with the entity. If the entity commits in a bundle, ClaimCenter checks this version number property in latest version in the database. ClaimCenter confirms that the cached original version of the entity is up to date. If the version numbers do not match, the current commit fails.

Record Locking for Concurrent Data Access

In addition to version protections, the system locks the database briefly during the commit. The server throws concurrent data exceptions if two different threads or two different servers in a cluster simultaneously modify the same entity at the exact same time.

User Interface Bundle Refreshes

In some cases, the application user interface automatically refreshes bundle entity data with the latest version from the database in cases that it seems appropriate. For example, changing from view-only to edit mode on data. There is no supported public API for you to programmatically refresh a bundle's entity data.

Details of What Happens During Bundle Commit

If ClaimCenter commits a bundle to help optimize database performance and application design, the following steps occur during bundle commit:

1. The application reserves a connection from the connection pool.
2. A bundle version number increments so the application can rollback to the existing state if something fails.

3. The application triggers pre-update rule sets.
4. The application attempts to refresh all entities that already existed but were **not** edited in the bundle. This ensures validation rules, which run soon, get the newest versions of the entities in this bundle.
5. The application triggers validation rule sets.
6. The application sets `editable` and `versionable` property values such as `updateTime`, `createTime`, and `user`.
7. The application increments the version number on the entity.
8. For any new entities (not yet in the database), the application creates internal IDs (the `entity.Id` property) for them and fixes any foreign key references to these new entities.
9. If any messaging destination was listening for relevant messaging events, the application triggers Event Fired rule sets once for each messaging destination. For example, if entities are added, changed, or removed, a messaging destination might listen for the *ENTITYNAME*`Added`, *ENTITYNAME*`Changed`, or *ENTITYNAME*`Removed` events. See “Messaging and Events” on page 139 in the *Integration Guide* for much more information on this topic.
10. The changed entities are computed.
11. The application writes all changed entities to the database connection.
12. Internally, the bundle is marked as written.
13. The application commits the database connection to the database, which either completely succeeds or fails.
14. The application updates the global cache of entities. This cache speeds up access to entities by caching recently used entities in memory on each server in an application cluster. By updating the cache, the current server sends messages to other servers in the cluster to remove cached entities that were just updated or deleted. Entities in their cache update to reflect the new unloaded state, but they are not immediately reread from the database on those servers. The other servers lazily reread entities from the database on those machines if necessary.

chapter 20

Gosu Templates

Gosu includes a native template system. Templates are text with embedded Gosu code within a larger block of text. The embedded Gosu code optionally can calculate a value and export the result as text in the location the code appears in the template text.

This topic includes:

Template Overview

Templates are text with embedded Gosu code within a larger block of text. The embedded Gosu code optionally can calculate a value and export the result as text in the location the code appears in the template text. There are two mechanisms to use Gosu templates:

- **Template syntax inside text literals.** Inside your Gosu code, use template syntax for an inline `String` literal values with embedded Gosu expressions. Gosu template syntax combines static text that you provide with dynamic Gosu code that executes at run time and returns a result. Gosu uses the result of the Gosu expression to output the dynamic output at run time as a `String` value.
- **Separate template files.** Define Gosu templates as separate files that you can execute from other code to perform actions and generate output. If you use separate template files, there are additional features you can use such as passing custom parameters to your template. For more details, see "Using Template Files" on page 293.

The simplest way to use to templates is to embed Gosu expressions that evaluate at run time and generate text in the place of the embedded Gosu expressions.

Template Expressions

Use the following syntax to embed a Gosu expression in `String` text:

```
${ EXPRESSION }
```

For example, suppose you want to display text with some calculation in the middle of the text:

```
var mycalc = 1 + 1
var s = "One plus one equals " + mycalc + "."
```

Instead of this multiple-line code, embed the calculation directly in the `String` as a template:

```
var s = "One plus one equals ${ 1 + 1 }."
```

If you print this variable, Gosu outputs:

```
One plus one equals 2.
```

Gosu runs your template expression at run time. The expression can include variables or dynamic calculations that return a value:

```
var s1 = "One plus one equals ${ myVariable }."
var s2 = "The total is ${ myVariable.calculateMyTotal() }."
```

At compile time, Gosu uses the built-in type checking system to ensure the embedded expression is valid and type safe.

If the expression does not return a value of type `String`, Gosu attempts to coerce the result to the type `String`.

Alternate Template Expression Syntax <%= … %>

The syntax `${ EXPRESSION }` is the preferred style for template expressions.

Additionally, Gosu provides an alternate template style. Use the three-character text `<%=` to begin the expression. Use the two-character text `%>` to end the expression. For example, you can rewrite the previous example as the following concise code:

```
var s = "One plus one equals <%= 1 + 1 %>."
```

Any surrounding text exports to the output directly.

When to Escape Special Characters for Templates

Gosu templates use standard characters in the template to indicate the beginning of a special block of Gosu code or other template structures. In some cases, to avoid ambiguity for the Gosu parser you must specially escape special characters.

For Non-Template-Tag Use, Escape ${ or <%

Gosu templates use the following two-character sequences to begin a template expression

- `${`
- `<%`

With a `String` literal in your code, if you want to use these to indicate template tags, do not need to escape these special characters.

If you want either of those two special two-character sequences actually in your `String` (not as a template tag), escape the first character of that sequence. To escape a character, add a backslash character immediately before it. For example:

- To define a variable containing the non-template text `"Hello${There"`:
  ```
  var s = "Hello\${There"
  ```
- To define a variable containing the non-template text `"Hello<%There"`:
  ```
  var s = "Hello\<%There"
  ```

If you use the initial character on its own (the next character would not indicate a special tag), you do not need to escape it. For example:

- To define a variable containing the non-template text `"Hello$There"`, simply use:
  ```
  var s = "Hello$There"
  ```
- To define a variable containing the non-template text `"Hello<There"`, simply use:
  ```
  var s = "Hello<There"
  ```

Within Template Tag Blocks, Use Standard Gosu Escaping Rules

In typical use, if you defined a `String`, you must escape it with the syntax `\"` to avoid ambiguity about whether you were ending the `String`. For example:

```
var quotedString = "\"This has double quotes around it\", is that correct?"
```

This creates a `String` with the following value, including quote signs:

```
"This has double quotes around it", is that correct?
```

However, if you use a template, this rule does not apply between your template-specific open and closing tags that contain Gosu code. Instead, use standard Gosu syntax for the code between those open and closing tags.

In other words, the following two lines are valid Gosu code:

```
var s = "${ "1" }"
var s = "${ "1" } \"and\" ${ "1" }"
```

Note that the first character within the template’s Gosu block is an unescaped quote sign.

However, the following is invalid due to improper escaping of internal double quotes:

```
var s = "${ \"1\" }"
```

In this invalid case, the first character within the template’s Gosu block is an escaped quote sign.

In the rare case that your Gosu code requires creating a `String` literal containing a quote character, remember that the standard Gosu syntax rules apply. This means that you will need to escape any double quote signs that are within the `String` literal. For example, the following is valid Gosu:

```
var quotedString = "${ "\"This has double quotes around it\", is that correct?" }"
```

Note that the first character within the template’s Gosu block is an unescaped quote sign. This template generates a `String` with the value:

```
"This has double quotes around it", is that correct?
```

IMPORTANT Be careful with how you escape double quote characters within your embedded Gosu code or other special template blocks.

Using Template Files

Instead of defining your templates in inline text, you can store a Gosu template as a separate file. Template files support all the features that inline templates support, as described in “Template Overview” on page 291. In addition, with template files you get additional advantages and features:

- **Separate your template definition from code that uses the template.** For example, define a template that generates a report or a notification email. You can then call this template from many places but define the template only once.
- **Encapsulate your template definition for better change control.** By defining the template in a separate file, your teams can edit and track template changes over time separate from code that uses the template.
- **Run Gosu statements (and return no value) using scriptlet syntax.** You can define one or more Gosu statements as a *statement list* embedded in the template. Contrast this with the template expression syntax described in “Template Overview” on page 291, which require Gosu *expressions* rather than Gosu *statements*. The result of scriptlet tags generate no output. For more information, see “Template Scriptlet Tags” on page 294.

- **Define template parameters.** Template files can define parameters that you pass to the template at run time. For more information, see “Template Parameters” on page 295.
- **Extend a template from a class to simplify static method calls.** If you call static methods on one main class in your template, you can simplify your template code using the `extends` feature. For more information, see “Extending a Template From a Class” on page 296.

Creating and Running a Template File

Gosu template files have the extension `.gst`. Create template files within the package hierarchy in the file system just as you create Gosu classes. Choose the package hierarchy carefully because you use this package name to access and run your template.

In your template file, include the template body with no surrounding quote marks. The following is a simple template:

```
One plus one equals ${ 1 + 1 }.
```

To create a new template within Studio, in the resource pane within the **Classes** section, right-click on a package. Next, right-click and choose **New** → **Template**.

The template is a first-class object in the Gosu type system within its package namespace. To run a template, get a reference to your template and call the `renderToString` method of the template.

For example, suppose you create a template file `NotifyAdminTemplate.gst` within the package `mycompany.templates`. Your fully-qualified name of the template is `mycompany.templates.NotifyAdminTemplate`.

Use the following code to render (run) your template:

```
var x = mycompany.templates.NotifyAdminTemplate.renderToString()
```

The variable `x` contains the `String` output of your template.

If you want to pass template parameters to your template, add additional parameters as arguments to the `renderToString` method. See “Template Parameters” on page 295 for details.

Output to a Writer

The `renderToString` method outputs the template results to a `String` value. Optionally, you can render the template directly to a Java writer object. Your writer must be an instance of `java.io.Writer`. To output to the writer, get a reference to the template and call its `render` method, passing the writer as an argument to the method.

For example, suppose you create a template file `NotifyAdminTemplate.gst` within the package `mycompany.templates`. If your variable `myWriter` contains an instance of `java.io.Writer`, the following Gosu statement renders the template to the writer:

```
mycompany.templates.NotifyAdminTemplate.render(myWriter)
```

If you use template parameters in your template, add your additional parameters after the writer argument. See “Template Parameters” on page 295 for details.

Template Scriptlet Tags

Text enclosed with the text `<%` and `%>` evaluate at run time in the order the parser encounters the text but generates nothing as output based on the result. These are called *scriptlet tags*. It is important to note that this type of tag has no equals sign in the opening tag. All plain text between scriptlet tags generate to the output within the scope and the logic of the scriptlet code.

The following simple template uses a scriptlet tag to run code to assign a variable and uses the result later:

```
<% var MyCalc = 1 + 2 %>One plus two is ${ MyCalc }
```

This prints the following:

```
One plus two is 3
```

It is important to note that the result of the scriptlet tag at the beginning of the template does **not** generate anything to the output. The value `3` exports to the result because later expression surrounded with the *expression* delimiters `${` and `}` instead of the scriptlet delimiters `<%` and `%>`.

The scope of the Gosu continues **across** scriptlet tags. Use this feature to write advanced logic that uses Gosu code that you spread across multiple scriptlet tags. For example, the following template code outputs “x is 5” if the variable x has the value 5, otherwise outputs `"x is not 5"`:

```
<% if (x == 5) { %>
x is 5
<% } else { %>
x is not 5
<% } %>
```

Notice that the `if` statement actually controls the flow of execution of later elements in the template. This feature allows you to control the export of static text in the template as well as template expressions.

Scriptlet tags are particularly useful when used with template parameters because you can define conditional logic as shown in the previous example. See “Template Parameters” on page 295 for details.

Use this syntax to iterate across lists, arrays, and other iterable objects. You can combine this syntax with the expression syntax to generate output from the inner part of your loop. Remember that the scriptlet syntax does not itself support generating output text.

For example, suppose you set a variable called `MyList` that contains a `List` of objects with a `Name` property. The following template iterates across the list:

```
<% for (var obj in MyList) {
  var theName = obj.Name %>
  Name: ${ theName }
<% } %>
```

This might generate output such as:

```
Name: John Smith
Name: Wendy Wheathers
Name: Alice Applegate
```

This example also shows a common design pattern for templates that need to combine complex logic in scriptlet syntax with generated text into the template within a loop:

1. Begin a template scriptlet (starting it with `<%`) to begin your loop.
2. Before ending the scriptlet, set up a variable with your data to export
3. End the scriptlet (closing it with `%>`).
4. Optionally, generate some static text
5. Insert a template expression to export your variable, surrounding a Gosu expression with `${` and `}` tags.
6. Add another template scriptlet (with `<%` and `%>`) to contain code that closes your loop. Remember that scriptlets share scope across all scriptlets in that file, so you can reference other variables or close loops or other Gosu structural elements.

IMPORTANT There is no supported API to generate template output from within a template scriptlet. Instead, design your template to combine template scriptlets and template expressions using the code pattern in this topic.

The scriptlet tags are available in template files, but not within `String` literals using template syntax.

Template Parameters

You can pass parameters of any type to your self-contained Gosu template files.

To support parameters in a template

1. Create a template file as described earlier in this topic.
2. At the top of the template, create a parameter definition with the following syntax:
   ```
   <%@ params(ARGLIST) %>
   ```
   In this case, *ARGLIST* is an argument list just as with a standard Gosu function. For example, the following argument list includes a `String` argument and a `boolean` argument:
   ```
   <%@ params(x : String, y : boolean) %>
   ```
3. Later in the template, use template tags that use the values of those parameters. You can use both the template expression syntax (`${` and `}`) and template scriptlet syntax (`<%` and `%>`). Remember that the expression syntax always returns a result and generates additional text. In contrast, the scriptlet syntax only executes Gosu statements.
4. To run the template, add your additional parameters to the call to the `renderToString` method or after the writer parameter to the `render` methods.

For example, suppose you create a template file `NotifyAdminTemplate.gst` within the package `mycompany.templates`. Edit the file to contain the following contents:

```
<%@ params(personName : String, contactHR: boolean) %>
The person ${ personName } must update their contact information in the company directory.

<% if (contactHR) { %>
Call the human resources department immediately.
<% } %>
```

In this example, the `if` statement (including its trailing curly brace) is within scriptlet tags. The `if` statement uses the parameter value at run time to conditionally run elements that appear later in the template. This template exports the warning to call the human resources department **only** if the `contactHR` parameter is `true`. Use if statements and other control statements to control the export of static text in the template as well as template expressions.

Run your template with the following code:

```
var x : String = mycompany.templates.NotifyAdminTemplate.renderToString("hello", true)
```

If you want to export to a character writer, use code like the following:

```
var x : String = mycompany.templates.NotifyAdminTemplate.render(myWriter, "hello", true)
```

For a ClaimCenter-specific example, suppose you want to display all claims associated with a policy. The corresponding template code with a list of claims parameter might look something like this:

```
<%@ params(claims : List<Claim>) %>
<% for (var thisClaim in claims) { %>
  Claim number: ${ thisClaim.ClaimNumber }
<% } %>
```

This template might generate something like:

```
  Claim number: HO-123456:C0001
  Claim number: HO-123456:C0002
  Claim number: HO-123456:C0003
```

You can use template parameters in template files, but not within `String` literals that use template syntax.

Extending a Template From a Class

Gosu provides a special syntax to simplify calling static methods on a class of your choosing. The metaphor for this template shortcut is that your template can *extend* from a type that you define. Technically, templates are not instantiated as objects. However, your template can call **static** methods on the specified class without fully-qualifying the class. Static methods are methods defined directly on a class, rather than on instances of the class. For more information, see “Modifiers” on page 178.

To use this feature, at the top of the template file, add a line with the following syntax:

```
<%@ extends CLASSNAME %>
```

`CLASSNAME` must be a fully-qualified class name. You cannot use a package name or hierarchy.

For example, suppose your template wants to clean up the email address with the `sanitizeEmailAddress` static method on the class `gw.api.email.EmailTemplate`. The following template takes one argument that is an email address:

```
<%@ params(address : String) %>
<%@ extends gw.api.email.EmailTemplate %>
Hello! The email address is ${sanitizeEmailAddress(address)}
```

Notice that the class name does **not** appear immediately before the call to the static method.

You can use the `extends` syntax in template files, but not within `String` literals that use template syntax.

Template Comments

You can add comments within your template. Template comments do not affect template output.

The syntax of a template comments is the following:

```
<%-- your comment here --%>
```

For example:

```
My name is <%-- this is a comment --%>John.
```

If you render this template file, it outputs:

```
My name is John.
```

You can use template comments in template files, but not within `String` literals that use template syntax.

Template Export Formats

Because HTML and XML are text-based formats, there is no fundamental difference between designing a template for HTML or XML export compared to a plain text file. The only difference is that the text file must conform to HTML and XML specifications.

HTML results must be a well-formed HTML, ensuring that all properties contain no characters that might invalidate the HTML specification, such as unescaped “<” or “&” characters. This is particularly relevant for especially user-entered text such as descriptions and notes.

Systems that process XML typically are **very strict** about syntax and well-formedness. Be careful not to generate text that might invalidate the XML or confuse the recipient. For instance, beware of unescaped “<” or “&” characters in a notes field. If possible, you could export data within an XML `<CDATA>` tag, which allows more types of characters and character strings without problems of unescaped characters.

chapter 21

Type System

Guidewire Gosu provides several ways to gather information about an object or other type. This ability for a programming language to query an object from the outside for this information is referred to as *reflection*. Use this information for debugging or to change program behavior based on information gathered at run time.

This topic includes:

Basic Type Coercion

Gosu uses the "`expression as TYPE`" construction to cast an expression to a specific type. This process is also called *coercion*.

Syntax

```
expression as TYPE
```

The expression must be compatible with the type. The following table lists the results of casting a simple numeric expression into one of the Gosu-supported data types. If you try to cast an expression to an inappropriate type, Gosu throws an exception.

Expression	Data type	Result or error
`(5 * 4) as Array`	*n/a*	`Type mismatch or possible invalid operator.` `java.lang.Object[] is not compatible with java.lang.Double`
`(5 * 4) as Boolean`	`Boolean`	`true`
`(5 * 4) as DateTime`	`DateTime`	`1969-12-31` (default value)
`(5 * 4) as Key`	*n/a*	`Type mismatch or possible invalid operator.` `com.guidewire.commons.entity.Key is not compatible with java.lang.Double`

Expression	Data type	Result or error
`(5 * 4) as String`	`String`	`20`
`(5 * 4) as Type`	*n/a*	`Type mismatch or possible invalid operator. MetaType:java.lang.Object is not compatible with java.lang.Double`

Why Use Coercion?

Gosu requires all variables to have types at compile type. All method calls and properties on objects must be correct with the compile time type.

If an object has a compile-time type that is higher in the type hierarchy (it is a supertype) than you need, coerce it to the appropriate specific type. This is required before accessing properties or methods on the object that are defined on a more specific type.

For example, when getting items from a list or array, the compile time type might be a supertype of the type you know that it is. For example, the compile time type is an `Object` but you know that it is always a more specific type due to your application logic. You must cast the item to the desired type before accessing properties and methods associated with the subtype you expect.

The following example coerces an `Object` to the type `MyClass` so the code can call `MyMethod` method. This example assumes `MyMethod` is a method on the class `MyClass`:

```
var objarray : Object[] = MyUtilities.GetMyObjectArray()

var o = objarray[0] // type of this variable is Object

var myresult = (o as MyClass).MyMethod()
```

Gosu provides automatic downcasting to simplify your code in `if` statements and similar structures. For more information, see “Automatic Downcasting for ‘typeis’ and ‘typeof’” on page 302.

For related information, see “Basic Type Checking” on page 300 and “Using Reflection” on page 304.

Basic Type Checking

Gosu uses the `typeis` operator to compare an expression’s type with a specified type. The result is always `Boolean`. A `typeis` expression cannot be fully determined at compile time. For example, at run time the expression may evaluate to a more specific subtype than the variable is declared as.

Typeis Syntax

```
OBJECT typeis TYPE
```

Typeis Examples

Expression	Result
`42 typeis Number`	`true`
`"auto" typeis String`	`true`
`person typeis Person`	`true`
`person typeis Company`	`false`

Similarly, you can use the `typeof` *`object`* operator to test against a specific type.

Typeof Syntax

```
typeof expression
```

Typeof Examples

Expression	Result
`typeof 42`	`Number`
`typeof "auto"`	`String`
`typeof (4 + 5)`	`Number`

In real-world code, typically you need to check an object against a type **or** its subtypes, not just a single type. In such cases, it is better to use `typeis` instead of `typeof`. The `typeof` keyword returns the exact type. If you test this value with simple equality with another type, it returns `false` if the object is a subtype.

For example, the following expression returns `true`:

```
"hello" typeis Object
```

In contrast, the following expression returns `false` because `String` is a subtype of `Object` but is a different type:

```
typeof "hello" == Object
```

If you want information from the type itself, you can access a type by name (typing its *type literal*) or use the `typeof` operator to get an object's type. For example:

```
var s = "hello"
var t = typeof s
```

In this example, the type of `s` is `String`, so the value of the `t` variable is now `String`.

Static Type ('statictypeof')

To get the compile-time type of an object and use it programmatically, use the `statictypeof` keyword. The result of an `statictypeof` expression does not vary at run time. Contrast this with the `typeof` keyword, which performs a run-time check of the object.

The following example illustrates this difference:

```
var i : Object = "hello"
print(typeof i)
print(statictypeof i)
```

This example prints the output:

```
java.lang.String
java.lang.Object
```

The variable is declared as `Object`. However, at run time it contains an object whose type is `String`, which is a subtype of `String`.

The following example also illustrates how this difference can affect `null` values and unexpected conditions:

```
var i : Boolean;
i = null;
print(typeof i)
print(statictypeof i)
```

This prints the output:

```
void
java.lang.Boolean
```

At run time, the value of `i` is `null`, so its type is `void`. However, the compile-time type of this variable is `Boolean`.

Is Assignable From

For advanced manipulation of type objects, including the method called `isAssignableFrom` that exists on types, see "Using Reflection" on page 304.

Even Types Have Types

All objects have types. This even applies to types (such as the type called `String`). The expression `typeof String` evaluates to a parameterized version of the type `Type`. Specifically:

```
Type<java.lang.String>
```

For advanced manipulation of the `Type` object, see "Using Reflection" on page 304.

Automatic Downcasting for 'typeis' and 'typeof'

To improve the readability of your Gosu code, Gosu automatically downcasts after a `typeis` expression if the type is a subtype of the original type. This is particularly valuable for `if` statements and similar Gosu structures. Within the Gosu code bounded by the `if` statement, you do not need to do casting (`as` *`TYPE`* expressions) to that subtype. Because Gosu confirms that the object has the more specific subtype, Gosu implicitly considers that variable's type to be the **subtype**, at least within that block of code.

For example, a common pattern for this feature looks like the following:

```
var VARIABLE_NAME : TYPE_NAME

if (VARIABLE_NAME typeis SUBTYPE_NAME) {

  // use the VARIABLE_NAME as SUBTYPE_NAME without casting
  // This assumes SUBTYPE_NAME is a subtype of TYPE_NAME
}
```

For example, the following example shows a variable declared as an `Object`, but downcasted to `String` within the `if` statement.

Because of downcasting, the following code is valid:

```
var x : Object = "nice"
var strlen = 0

if( x typeis String ) {
  strlen = x.length
}
```

It is important to note that `length` is a property on `String`, not `Object`. The downcasting from `Object` to `String` means that you do not need an additional casting around the variable `x`. In other words, the following code is equivalent but has an **unnecessary** cast:

```
var x : Object = "nice"
var strlen = 0

if( x typeis String ) {
  strlen = (x as String).length // "length" is a property on String, not Object
}
```

Do not write Gosu code with unnecessary casts. Use automatic downcasting to write easy-to-read and concise Gosu code.

The automatic downcasting happens for the following types of statements;

- `if` statements. For more information, see "If() … Else() Statements" on page 86.
- `switch` statements. For more information, see "Switch() Statements" on page 89. For example:

```
uses java.util.Date

var x : Object = "neat"
switch( typeof( x ) ){
  case String :
    print( x.charAt( 0 ) ) // without automatic downcasting, this method call fails without casting
    break
  case Date :
    print( x.Time ) // without automatic downcasting, this property access fails without casting
    break
}
```

- ternary conditional expression, such as "`x typeis String ? x.length : 0`". Downcasting only happens in the part of the expression that corresponds to it being true (the first part). For more information, see "Conditional Ternary Expressions" on page 76.

This automatic downcasting works when the item to the left of the `typeis` keyword is a symbol or an entity path, but not on other expressions.

There are a several situations that cancel the `typeis` inference:

- Reaching the end of the extent of the scope for which inference is appropriate. In other words:
  - The end of an `if` statement
  - The end of a `switch` statement
  - The end of a ternary conditional expression in its `true` clause
- Assigning any value to the symbol (the variable) you checked with `typeis` or `typeof`. This applies to `if` and `switch` statements.
- Assigning any value to any part of an entity path you checked with `typeis` or `typeof`. This applies to `if` and `switch` statements.
- An `or` keyword in a logical expression
- The end of an expression negated with the `not` keyword
- In a `switch` statement, a `case` section does not use automatic downcasting if the previous case section is unterminated by a `break` statement. For example, the following Gosu code us valid and both `case` sections using automatic downcasting:

```
uses java.util.Date

var x : Object = "neat"
switch( typeof( x ) ){
  case String :
    print( x.charAt( 0 ) ) // without automatic downcasting, this method call fails without casting
    break
  case Date :
    print( x.Time ) // without automatic downcasting, this property access fails without casting
    break
}
```

  However, Gosu allows you to remove the first `break` statement. Removing a break statement allows the execution to fall through to the next `case` section. In other words, if the type is `String`, Gosu runs the `print` statement in the `String` case section. Next, Gosu run statements in the next `case` section also. This does not change the type system behavior of the section whose `break` statement is now gone (the first section). However, there is no downcasting for the following `case` section since two different cases share that series of Gosu statements. The compile time type of the switched object reverts to the compile-time type of that variable at the beginning of the `switch` statement.

  For example, the following code has a compile error because it relies on downcasting.

```
uses java.util.Date

var x : Object = "neat"
switch( typeof( x ) ){
  case String :
    print( x.charAt( 0 ) ) // without automatic downcasting, this method call fails without casting
  case Date :
    print( x.Time ) // COMPILE ERROR. The compile time type reverts to Object (no Time property!)
    break
}
```

  To work around this problem, remember that the compile time type of the switched object reverts to whatever the compile-time type is before the `switch` statement. Simply cast the variable with the `as` keyword before accessing type-specific methods or properties. For example:

```
uses java.util.Date

var x : Object = "neat"
switch( typeof( x ) ){
  case String :
    print( x.charAt( 0 ) ) // without automatic downcasting, this method call fails without casting
  case Date :
    print( (x as Date).Time ) // this is now valid Gosu code
    break
}
```

Using Reflection

Once you know what type something is, you can use reflection to learn about the type. Although each `Type` object itself has properties and methods on it, the most interesting properties and methods are on the *type*.`TypeInfo` object. For example, you can get a type's complete set of properties and methods at run time by getting the `TypeInfo` object.

> **WARNING** In general, avoid using reflection to get properties or call methods. In almost all cases, you can write Gosu code to avoid reflection. Using reflection dramatically limits how Gosu and Guidewire Studio can alert you to serious problems at compile time. In general, it is better to detect errors at compile time rather than unexpected behavior at run time. Only use reflection if there is no other way to do what you need.

The following example shows two different approaches for getting the `Name` property from a type:

```
print(Integer.Name)     // directly from a Type
print((typeof 29).Name) // getting the Type of something
```

This prints:

```
java.lang.Integer
int
```

Get Properties Using Reflection

The *type*.`TypeInfo` object includes a property called properties, that contains a list of type properties.

Each item in that list include metadata properties such as for the name (`Name`) and a short description (`ShortDescription`).

For example, paste the following code into the Gosu Tester:

```
var object = "this is a string"
var s = ""
var props =  (typeof object).TypeInfo.Properties

for (m in props) {
  s = s + m.Name + "  "
}

print(s)
```

This code prints something similar to the following:

```
Class  itype  Bytes  Empty  CASE_INSENSITIVE_ORDER  length  size  HasContent
NotBlank  Alpha  AlphaSpace  Alphanumeric  AlphanumericSpace  Numeric  NumericSpace  Whitespace
```

You can also call properties using reflection using the square bracket syntax, similar to using arrays. For example, paste the following code into the Gosu Tester:

```
// get the CURRENT time
var s = new DateTime()

// createa  String containing a property name
var propName = "hour"

// get a property name using reflection
print(s[propName])
```

If the time is currently 5 PM, this code prints:

```
5
```

Get Methods Using Reflection

Paste the following code into the Gosu Tester:

```
var object = "this is a string"
var s = ""
var methods =  (typeof object).TypeInfo.Methods

for (m in methods) {
```

```
  s = s + m.Name + "  "
}

print(s)
```

This code prints code that looks like this (truncated for clarity):

```
wait()  wait( long, int )  wait( long )  hashCode()  getClass()  equals( java.lang.Object )
toString()  notify()  notifyAll()  @itype()  compareTo( java.lang.String )  charAt( int )
length()  subSequence( int, int )  indexOf( java.lang.String, int )  indexOf( java.lang.String )
indexOf( int )  indexOf( int, int )  codePointAt( int )  codePointBefore( int )
```

You can also get information about individual methods. You can even call methods by name (given a `String` for the method name) and pass a list of parameters as `object` values. You can call a method using the method's `CallHandler` property, which contains a `handleCall` method.

The following example gets a method by name and then calls it. This example uses the `String` class and its `compareTo` method, which returns 0, 1, or -1. Paste the following code into the Gosu Tester

```
var mm = String.TypeInfo.Methods
var myMethodName = "compareTo"

// find a specific method by name using "collections" and "blocks" features...
var m = mm.findFirst( \ i -> i.Name == myMethodName )

print("Name is " + m.Name)
print("Number of parameters is " + m.Parameters.length)
print("Name of first parameter is " + m.Parameters[0].DisplayName)
print("Type of first parameter is " + m.Parameters[0].IntrinsicType)

// set up an object whose method to call. in this case, use a String
var obj = "a"
var comparisonString = "b"

// call the method using reflection! ** note: this returns -1 because "a" comes before "b"
print(m.CallHandler.handleCall( obj, { comparisonString } ))

// in this example, this was equivalent to the code:
print(obj.compareTo(comparisonString))
```

This code prints:

```
Name is compareTo
Number of parameters is 1
Name of first parameter is String
Type of first parameter is java.lang.String
-1
-1
```

Compare Types Using Reflection

You can compare the type of two objects in several ways.

You can use the equality (==) operator to test types. However, the equality operator is almost always inappropriate because it returns `true` only for exact type matches. It returns `false` if one type is a subtype of the other or if the types are in different packages.

Instead, use the `*type*.isAssignableFrom(*otherType*)` method to determine whether the types are compatible for assignment. This method considers the possibility of subtypes (such as subclasses) in a way that the equality operator does not. The method determines if the type argument is either the same as, or a superclass of (or super-interface of) the type.

The `*sourceType*.isAssignableFrom(*destinationType*)` method looks only at the supertypes of the source type. Although Gosu statements can assign a value of one **unrelated** type to another using coercion, the `isAssignableFrom` method always returns `false` if coercion of the data would be necessary. For example, Gosu can convert `boolean` to `String` or from `String` to `boolean` using coercion, but `isAssignableFrom` method returns `false` for those cases.

Gosu provides a variant of this functionality with the Gosu `typeis` operator. Whereas `*type*.isAssignableFrom(...)` operates between a type and another type, the `typeis` operates between an object and a type.

Paste the following code into the Gosu Tester:

For example, paste the following code into the Gosu Tester:

```
var s : String = "hello"
var b : Boolean = true

print("Typeof s: " + (typeof s).Name)
print("Boolean assignable from String : " + (typeof s).isAssignableFrom((typeof b)))
print("true typeis String: " + (b typeis String))
print("Object assignable from String: " + (Object).isAssignableFrom( String ))
print("Compare a string to object using typeis: " + (s typeis Object))

// Using == to compare types is a bad approach if you want to check for valid subtypes...
print("Compare a string to object using == : " + ((typeof s) == Object))
```

This code prints:

```
Typeof s: java.lang.String
Boolean assignable from String : false
true typeis String: false
Object assignable from String: true
Compare a string to object using typeis: true
Compare a string to object using == : false
```

Type Object Properties

The `Type` type is a metatype, which means that it is the type of all types. There are various methods and properties that appear directly on the type `Type` and all are supported.

Refer to the Gosu API Reference in the Studio Help menu for the full reference for all properties and methods.

The `Type` type includes the following important properties:

Property	Description
`Name`	The human-readable name of this type.
`TypeInfo`	Properties and methods of this type. See “Basic Type Checking” on page 300 for more information and examples that use this `TypeInfo` object.
`SuperType`	The supertype of this type, or `null` if there is no supertype.
`IsAbstract`	If the type is abstract, returns `true`. See “Modifiers” on page 178.
`IsArray`	If the type is an array, returns `true`.
`IsFinal`	If the type is final, returns `true`. See “Modifiers” on page 178.
`IsGeneric`	If the type is generic, returns `true`. See “Gosu Generics”, on page 221.
`IsInterface`	If the type is an interface, returns `true`. See “Interfaces”, on page 191.
`IsParameterized`	If the type is parameterized, returns `true`. See “Gosu Generics”, on page 221.
`IsPrimitive`	If the type is primitive, returns `true`.

For more information about the `isAssignableFrom` method on the `Type` object, refer to the previous section.

Working with Primitive Types

In Gosu, primitive types such as `int` and `boolean` exist primarily for compatibility with the Java language. Gosu uses these Java primitive types to support extending Java classes and implementing Java interfaces. From a Gosu language perspective, primitives are different only in subtle ways from object-based types such as `Integer` and `Boolean`. Primitive types can be automatically coerced (converted) to non-primitive versions or back again by the Gosu language in almost all cases. For example, from `int` to `Integer` or from `Boolean` to `boolean`.

You typically do **not** need to know the differences, and internally they are stored in the same type of memory location so there is no performance benefit to using primitives. Internally, primitives are stored as objects, and there is no speed improvement for using Gosu language primitives instead of their boxed versions, such as `int` compared to `Integer`.

The `boolean` type is a primitive, sometimes called an *unboxed type*. In contrast, `Boolean` is a class so `Boolean` is called a *boxed type* version of the `boolean` primitive. A boxed type is basically a primitive type wrapped in a shell of a class. These are useful so that code can make assumptions about all values having a common ancestor type `Object`, which is the root class of all class instances. For example, collection APIs require all objects to be of type `Object`. Thus, collections can contain `Integer` and `Boolean`, but not the primitives `int` or `boolean`.

However, there are differences while handling uninitialized values, because variables declared of a primitive type cannot hold the `null` value, but regular `Object` variable values can contain `null`.

For example, paste the following code into the Gosu Tester:

```
var unboxed : boolean = null  // boolean is a primitive type
var boxed : Boolean = null    // Boolean is an Object type, a non-primitive

print(unboxed)
print(boxed)
```

This code prints:

```
false
null
```

These differences are also notable if you pass primitives to `isAssignableFrom`. This method only looks at the type hierarchy and returns `false` if comparing primitives.

For example, paste the following code into the Gosu Tester:

```
var unboxed : boolean = true  // boolean is a primitive type
var boxed : Boolean = true    // Boolean is an Object type, a non-primitive

print((typeof boxed).IsPrimitive)
print((typeof unboxed).IsPrimitive)
print((typeof unboxed).isAssignableFrom( (typeof boxed) ))
```

This code prints:

```
false
true
false
```

In Gosu, the boxed versions of primitives use the Java versions. Because of this, in Gosu you find them defined in the `java.lang` package. For example, `java.lang.Integer`.

For more information about `Boolean` and `boolean`, see “Boolean” on page 44.

Java Type Reflection

Gosu implements a dynamic type system that is designed to be extended beyond its native objects. Do not confuse this with being *dynamically typed* because Gosu is *statically typed*. Gosu’s dynamic type system enables Gosu to work with a variety of different types used in Guidewire applications.

These types include Gosu classes, Java classes, business entity objects, typelists, XML types, SOAP types, and other types. These different types plug into Gosu’s type system in a way similar to how Gosu business entities connect to the type system.

In almost all ways, Gosu does not care about the difference between a Java class or a native Gosu object. They are all exposed to the language through the same abstract type system so you can use Java types directly in your code. You can even extend Java classes, meaning that you can write Gosu types that are subtypes of Java types. Similarly, you can implement or even extend Java interfaces from Gosu.

The Gosu language transparently exposes and uses Java classes as Gosu objects through the use of Java `BeanInfo` objects. Java `BeanInfo` objects are analogous to Gosu’s `TypeInfo` information. They both encapsulate type metadata, including properties and methods on that type. All Java classes have `BeanInfo` information either explicitly provided with the Java class or can be dynamically constructed at runtime. Gosu examines a Java class’s `BeanInfo` and determines how to expose this type to Gosu. Because of this, your Gosu code can use the Gosu reflection APIs discussed earlier in this section with Java types.

Note: For a related topic, see “Java and Gosu”, on page 101.

Type System Class

You can use the class `gw.lang.reflect.TypeSystem` for additional supported APIs for advanced type system introspection. For example, its `getByFullName` method can return a `Type` object from a `String` containing its fully-qualified name.

For example, the following code gets a type by a `String` version of its fully-qualified name and instantiates it using the type information for the type:

```
var myFullClassName = "com.mycompany.MyType"
var type = TypeSystem.getByFullName( myFullClassName )
var instance = type.TypeInfo.getConstructor( null ).Constructor.newInstance( null )
```

Refer to the Gosu API Reference in Studio in the **Help** menu for details of additional methods on this class.

Compound Types

To implement some other features, Gosu supports a special kind of type called a *compound type*. A compound type combines one base class and additional interfaces that the type supports. You can declare a variable to have a compound type. However, typical usage is only when Gosu automatically creates a variable with a compound type.

For example, suppose you use the following code to initialize list values:

```
var x : List<String> = {"a", "b", "c"}
```

Note: The angle bracket notation indicates support for parameterized types, using Gosu generics features. For more information, refer to “Gosu Generics” on page 221.

You could also use this syntax using the `new` operator:

```
var x = new List<String>(){"a", "b", "c"}
```

Gosu also supports an extremely compact notation that does not explicitly include the type of the variable:

```
var x = {"a", "b", "c"}
```

It might surprise you that this last example is valid Gosu and is typesafe. Gosu infers the type of the `List` to be the least upper bound of the components of the list. In the simple case above, the type of the variable `x` at compile time is `List<String>`. If you pass different types of objects, Gosu finds the most specific type that includes all of the items in the list.

If the types implement interfaces, Gosu attempts to preserve the commonality of interface support in the list type. This means your list acts as expected with APIs that rely on support for the interface. In some cases, the resulting type is a *compound type*, which combines the following into a single type:

- **at most** one *class*
- one or more *interfaces*

For example, the following code initializes an `int` and a `double`:

```
var s  = {0, 3.4}
```

The resulting type of `s` is `ArrayList<java.lang.Comparable & java.lang.Number>`. This means that it is an array list of the compound type of the class `Number` and the interface `Comparable`.

Note: The `Number` class does not implement the interface `Comparable`. If it did, then the type of `s` would simply be `ArrayList<java.lang.Number>`. However, since it does not implement that interface, but both `int` and `double` implement that interface, Gosu assigns the compound type that includes the interfaces that they have in common.

This new compound type with type inference works with maps, as shown in the following examples:

```
var numbers = {0 -> 1, 3 -> 3.4}
var strings = {"a" -> "value"}
```

This also works with sets, as shown in the following example:

```
var s : Set = {1,2,3}
```

Compound Types using Composition (Delegates)

Gosu also creates compound types in the special case of using the delegate keyword with multiple interfaces. For more information, see “Using Gosu Composition” on page 195.

chapter 22

Concurrency

This topic describes Gosu APIs that protect shared data from access from multiple threads.

This topic includes:

Overview of Thread Safety and Concurrency

If more than one Gosu thread interacts with data structures that another thread needs, you must ensure that you protect data access to avoid data corruption. Because this topic involves concurrent access from multiple threads, this issue is generally called *concurrency*. If you design your code to safely get or set concurrently-accessed data, your code is called *thread safe*.

The most common situation that requires proper concurrency handling is data in class static variables. Static variables are variables that are stored once per class rather than once per instance of the class. If multiple threads on the same Java virtual machine access this class, you must ensure that any simultaneous access to this data safely gets or sets this data.

If you are experienced with multi-threaded programming and you are certain that static variables or other shared data is necessary, you must ensure that you *synchronize* access to static variables. Synchronization refers to locking access between threads to shared resources such as static variables.

There are other special cases in which you must be particularly careful. For example, if you want to manage a single local memory cache that applies to multiple threads, you must carefully synchronize all reads and writes to shared data.

In a Guidewire application, some contexts always require proper synchronization. For example, in a plugin implementation exactly one instance of that plugin exists in the Java virtual machine on each server. This means the following:

- Your plugin must support multiple simultaneous calls to the same plugin method from different threads. You must ensure multiple calls to the plugin never access the same shared data. Alternatively, protect access to shared resources so that two threads never access it simultaneously.
- Your code must support multiple simultaneous calls to a plugin instance. For example, ClaimCenter might call two different plugin methods at the same time. You must ensure multiple method calls to the plugin never access the same shared data. Alternatively, protect access to shared resources so that two threads never actually access it simultaneously.
- Your plugin implementation must support multiple user sessions. Generally speaking, do not assume shared data or temporary storage is unique to one user request (one HTTP request of a single user).

> **WARNING** Static variables can be extremely dangerous in a multi-threaded environment. Using static variables in a multi-threaded environment can cause problems in a production deployment if you do not properly synchronize access. If such problems occur, they are extremely difficult to diagnose and debug. Timing in an multi-user multi-threaded environment is difficult, if not impossible, to control in a testing environment.

Gosu provides the following types of concurrency APIs to make it easy for you to write thread-safe code:

- **Scoping classes (pre-scoped maps).** Scope-related utilities in the class `gw.api.web.Scopes` help synchronize and protect access to shared data. These APIs return `Map` objects into which you can get and put data using different scope semantics. Gosu automatically synchronizes the `Map` objects to provide proper concurrent access semantics. For more information, see “Gosu Scoping Classes (Pre-scoped Maps)” on page 313
- **Lazy concurrent variables.** The `LazyVar` class (in `gw.util.concurrent)` implements what some people call a *lazy variable*. This means Gosu constructs it only the first time some code uses it. Because the `LazyVar` class uses the Java concurrency libraries, access to the lazy variable is thread-safe. The `LazyVar` class wraps the double-checked locking pattern in a typesafe holder. For more information, see “Concurrent Lazy Variables” on page 314
- **Concurrent cache.** The `Cache` class (in `gw.util.concurrent)` declares a cache of values you can look up quickly and in a thread-safe way. It declares a concurrent cache similar to a Least Recently Used (LRU) cache. Because the `Cache` class uses the Java concurrency libraries, access to the concurrent cache is thread-safe. For more information, see “Concurrent Cache” on page 314.

> **WARNING** Caches are difficult to implement and use, and caches can cause subtle problems. Use caches only as a last result for performance. If you use a cache, Guidewire strongly recommends multiple people on your team carefully review any cache-related code.

- **Support for Java monitor locks, reentrant locks, and custom reentrant objects.** Gosu provides access to Java-based classes for monitor locks and reentrant locks in the Java package `java.util.concurrent`. Gosu makes it easier to access these classes with easy-to-read `using` clauses that also properly handle cleanup if exceptions occur. Additionally, Gosu makes it easy to create custom Gosu objects that support an easy-to-read syntax for reentrant object handling. For more information, see “Concurrency with Monitor Locks and Reentrant Objects” on page 316.

Concurrency APIs Do Not Synchronize Across Clusters

None of these concurrency APIs affect shared data across clusters. In practice, the only data shared across clusters is entity data. ClaimCenter includes built-in systems to synchronize and intelligently cache entity data across a cluster. For more information about how ClaimCenter manages access to collections of entities, see “Bundles and Transactions” on page 275.

Do Not Attempt to Synchronize Guidewire Entities

Never try to synchronize access to Guidewire entities defined in the data model configuration files. ClaimCenter automatically manages synchronized access and entity loading and caching, both locally and across a cluster (if you use clusters). For more information about how ClaimCenter manages access to collections of entities, see "Bundles and Transactions" on page 275.

Gosu Scoping Classes (Pre-scoped Maps)

Gosu provides scope-related utility methods in the class `gw.api.web.Scopes`. These static methods help synchronize and protect access to shared data using synchronized `Map` objects that follow standard web-application scoping semantics.

IMPORTANT These methods are available only in execution contexts that are associated with a web request. If you attempt to accessed these methods in other contexts, Gosu throws an `IllegalStateException` exception. Be aware that how this data is stored is dependent on the application server container in which your application runs. Your data must satisfy any constraints that container imposes. For example, some application containers might require that your objects are serializable (implement the `Serializable` interface).

Call methods that correspond to different scopes:

Scope	Meaning	Method	Description
Request scope	A single thread-local request.	`getRequest`	Returns a `Map` to store and retrieve values whose lifespan is the lifespan of the request. This map is not synchronized since multiple threads typically cannot get to the same request object. You could create unexpected situations by passing this object to other threads, so you must avoid such actions.
Session	One web session	`getSession`	Returns a `Map` to store and retrieve values whose lifespan is the lifespan of the users session. This map is automatically synchronized since multiple threads can access the session simultaneously. For example, web AJAX requests.
Application	The entire ClaimCenter application, including all requests and subthreads.	`getApplication`	Returns a `Map` to store and retrieve values whose lifespan is the lifespan of the web application. This is almost identical to static variables, but Gosu clears the map if a servlet shuts down and is later restarted.

For example, the following Gosu class creates an application-scoped variable.

```
class MyClass {

  // lazy variable using a block that calls a resource-intensive operation that retuns a String
  static var _data : java.util.Map

  construct() {

    // create an instance of a thread-safe shared Map with application scope
    _data = gw.api.web.Scopes.getApplication()

   // set variable in our scoped object. The object ensures any access is thread-safe.
    _data["Name"] = "John Smith"
  }
}
```

Concurrent Lazy Variables

In addition to using the Java native concurrency classes, Gosu includes utility classes that add additional concurrency functionality. The `LazyVar` class implements what some people call a *lazy variable*. This means Gosu constructs it only the first time some code uses it. Because the `LazyVar` class uses the Java concurrency libraries, access to the lazy variable is thread-safe. The `LazyVar` class wraps the double-checked locking pattern in a type-safe holder.

In Gosu, you will see the `make` method signature `LazyVar.make(gw.util.concurrent.LazyVar.LazyVarInit)` method signature, which returns the lazy variable object. This method requires a Gosu block that creates an object. Gosu runs this block on the first **access** of the `LazyVar` value. An example is easier to understand than the method signature. The following example passes a block as an argument to `LazyVar.make(...)`. That block creates a new `ArrayList` parameterized to the `String` class:

```
var _lazy = LazyVar.make( \-> new ArrayList<String>() )
```

As you can see, the parameter is a block that creates a new object. In this case, it returns a new `ArrayList`. You can create any object. In real world cases this block might be very resource-intensive to create (or load) this object.

It is best to let Gosu infer the proper type of the block type or the result of the make method, as shown in this example. This simplifies your code so that you do not need to use explicit Gosu generics syntax to define the type of the block itself, such as the following verbose version:

```
var _lazy : LazyVar<List<String>> = LazyVar.make( \-> new ArrayList<String>() )
```

To use the lazy variable, just call its `get` method:

```
var i = _lazy.get()
```

If the Gosu has not yet run the block, it does when you access it. If Gosu has run the block, it simply returns the cached value and does not rerun the block.

A good approach to using lazy variables is to define it as a static variable and then define a property accessor function to abstract away the implementation of the variable. The following is an example inside a Gosu class definition:

```
class MyClass {

  // lazy variable using a block that calls a resource-intensive operation that retuns a String
  var _lazy = LazyVar.make( \-> veryExpensiveMethodThatRetunsAString() )

  // define a property get function that gets this value
  property get MyLazyString() : String {
    return _lazy.get()
  }
}
```

If any code accesses the property `MyLazyString`, Gosu calls its property accessor function. The property accessor always calls the get method on the object. However, Gosu only runs the very expensive method once, the first time someone accesses the lazy variable value. If any code accesses this property again, the cached value is used. Gosu does not execute the block again. This is useful in cases where you want some system to come up quickly and only pay incremental costs for resource-intensive value calculations.

Concurrent Cache

A similar class to the `LazyVar` class is the `Cache` class. It declares a cache of values you can look up quickly and in a thread-safe way. It declares a concurrent cache similar to a Least Recently Used (LRU) cache. Because the `Cache` class uses the Java concurrency libraries, access to the cache is thread-safe.

To create a thread-safe cache

1. Decide the key and value types for your cache based on input data. For example, perhaps you want to pass a `String` and get an `Integer` back from the cache.
2. Use the key and value types to parameterize the Cache type using Gosu generics syntax. For example, if you want to pass a `String` and get an `Integer` back from the cache, create a new `Cache<String, Integer>`.
3. In the constructor, pass the following arguments:
   - a name for your cache as a `String` - the implementation uses this name to generate logging for cache misses
   - the size of your cache, as a number of slots
   - a block that defines a function that calculates a value from an input value. Presumably this is a resource-intensive calculation.

   For example,

   ```
   // A cache of string values to their upper case values
   var myCache = new Cache<String, String>( "My Uppercase Cache", 100, \ s -> s.toUpperCase() )
   ```
4. To use the cache, just call the get method and pass the input value (the key). If the value is in the cache, it simply returns it from the cache. If it is not cached, Gosu calls the block and calculates it from the input value (the key) and then caches the result. For example:

   ```
   print(myCache.get("Hello world")
   print(myCache.get("Hello world")
   ```

   This prints:

   ```
   "HELLO WORLD"
   "HELLO WORLD"
   ```

   In this example, the first time you call the `get` method, it calls the block to generate the upper case value. The second time you call the `get` method, the value is the same but Gosu uses the cached value. Any times you call the `get` method later, the value is the same but Gosu uses the cached value, assuming it still in the cache. If too many items were added to the cache and your desired item is unavailable, Gosu reruns the block to regenerate the value. Gosu then caches the result.

Alternatively, if you want to use a cache within some other class, you can define a static instance of the cache. The static variable definition itself defines your block. Again, because the `Cache` class uses the Java concurrency libraries, it is thread-safe. For example, in your class definition, define a static variable like this:

```
static var _upperCaseCache = new Cache<Foo, Bar>( 1000, \ foo -> getBar( foo ) )
```

To use your cache, your class can get a value from the cache using code like the following. In this example, `inputString` is a `String` variable that may or may not contain a `String` that you used before with this cache:

```
var fastValue = _upperCaseCache.get( inputString )
```

The first time you call the `get` method, it calls the block to generate the upper case value.

Any later times you call the `get` method, the value is the same but Gosu uses the cached value, assuming it still in the cache. If too many items were added to the cache and your desired item is unavailable, Gosu reruns the block to regenerate the value. Gosu then caches the result in the concurrent cache object.

An even better way to use the cache is to abstract the cache implementation into a property accessor function. Let the private static object `Cache` object (as shown in the previous example) handle the actual cache. For example, define a property accessor function such as:

```
static property get function UpperCaseQuickly( str : String ) {
    return _upperCaseCache.get( str )
  }
```

These are demonstrations only with a simple and non-resource-intensive operation in the block. Generally speaking, it is only worth the overhead of maintaining the cache if your calculation is resource-intensive combined with potentially repeated access with the same input values.

WARNING Caching can be difficult and error prone in complex applications. It can lead to run time errors and data corruption if you do not do it carefully. Only use caches as a last resort for performance issues. Because of the complexity of cache code, always have multiple experienced programmers review any cache-related code.

Concurrency with Monitor Locks and Reentrant Objects

From Gosu, you can use the Java 1.5 concurrency classes in the package `java.util.concurrent` to synchronize the variable's data to prevent simultaneous access to the data.

The simplest form is to define a static variable for a lock in your class definition. Next, define a property get accessor function that uses the lock and calls another method that performs the task you must synchronize. This approach uses a Gosu `using` clause with reentrant objects to simplify concurrent access to shared data.

For example:

```
...

// in your class definition, define a static variable lock
static var _lock = new ReentrantLock()

// a property get function uses the lock and calls another method for the main work
property get SomeProp() : Object
  using( _lock ) {
    return _someVar.someMethod() // do your main work here and Gosu synchronizes it
  }

...
```

The `using` statement automatically cleans up the lock, even if there code throws exceptions.

In contrast, this is a traditionally-structured verbose use of a lock using `try` and `finally` statements:

```
uses java.util.concurrent

...

static var _lock = new ReentrantLock()
  static var _someVar = ...

  property get SomeProp() : Object {
    _lock.lock()
    try {
      return _someVar.someMethod()
    } finally {
      _lock.unlock()
    }
  }
```

Alternatively, you can do your changes within Gosu blocks:

```
uses java.util.concurrent

...

property get SomeProp() : Object {
  var retValue : Object
  _lock.with( \-> {
    retValue = _someVar.someMethod()
  })
  return retValue
```

```
}
```

Note: Although this approach is possible, returning the value from a block imposes some more restrictions on how you implement your `return` statements. Instead, it is usually better to use the `using` statement structure at the beginning of this topic.

The using statement version works with these lock objects because Gosu considers this objects *reentrant*.

Re-entrant objects are objects that help manage safe access to data that is shared by re-entrant or concurrent code execution. For example, if you must store data that is shared by multiple threads, ensure that you protect against concurrent access from multiple threads to prevent data corruption. The most prominent type of shared data is class *static variables*, which are variables that are stored on the Gosu class itself.

For Gosu to recognize a valid reentrant object, the object must have one of the following attributes:

- Implements the `java.util.concurrent.locks.Lock` interface. This includes the Java classes in that pacakge: `ReentrantLock`, `ReadWriteLock`, `Condition`.
- Casted to the Gosu interface `IMonitorLock`. You can cast **any** arbitrary object to `IMonitorLock`. This is useful to cast Java monitor locks to this Gosu interface. For more information about monitor locks, refer to:
  `http://en.wikipedia.org/wiki/Monitor_(synchronization)`
- Implements the Gosu class `gw.lang.IReentrant`. This interface contains two methods with no arguments: `enter` and `exit`. Your code must properly lock or synchronize data access as appropriate during the `enter` method and release any locks in the `exit` method.

For blocks of code using locks (code that implements `java.util.concurrent.locks.Lock`), a `using` clause simplifies your code.

The following code uses the `java.util.concurrent.locks.ReentrantLock` class using a longer (non-recommended) form:

```
// in your class variable definitions...
var _lock : ReentrantLock = new ReentrantLock()


function useReentrantLockOld() {
  _lock.lock()
  try {
      // do your main work here
    }
  finally {
    _lock.unlock()
  }
}
```

In contrast, you can write more readable Gosu code using the `using` keyword:

```
// in your class variable definitions...
var _lock : ReentrantLock = new ReentrantLock()


function useReentrantLockNew() {
  using( _lock ) {
    // do your main work here
  }
}
```

Similarly, you can cast any object to a monitor lock by adding "`as IMonitorLock`" after the object. For example, the following method call code uses itself (using the special keyword `this`) as the monitor lock:

```
function monitorLock() {
  using( this as IMonitorLock ) {
      // do stuff
    }
  }
```

This approach effectively is equivalent to a `synchronized` block in the Java language.

Assigning Variables Inside 'using' Expression Declaration

The using clause supports assigning a variable inside the declaration of the `using` clause.

This is useful if the expression that you pass to the `using` expression is both:

- something other than a single variable
- you want to reference it from inside the statement list inside the `using` clause declaration

For example, suppose you call a method that returns a file handle and you pass that to the `using` clause as the lock. From within the `using` clause contents, you probably want to access the file so you can iterate across its contents.

To simplify this kind of code, assign the variable before the expression using the `var` keyword:

```
using ( var VARIABLE_NAME = EXPRESSION ) {
  // code that references the VARIABLE_NAME variable
}
```

For example:

```
using( var out = new FileOutputStream( this, false ) ) {
  out.write( content )
}
```

Passing Multiple Items to the ‘using’ Statement

You can pass multiple items in the using clause expression. Separate each item by a comma character.

For example,

```
function useReentrantLockNew() {
  using( _lock1, _lock2, _lock3) {
    // do your main work here
  }
}
```

You can combine the multiple item feature with the ability to assign variables. For more about assigning variables, see “Assigning Variables Inside ‘using’ Expression Declaration” on page 317 .

For example:

```
using( var lfc = new FileInputStream(this).Channel,
       var rfc = new FileInputStream(that).Channel ) {

      var lbuff  = ByteBuffer.allocate(bufferSize)
      var rbuff  = ByteBuffer.allocate(bufferSize)

      while (lfc.position() < lfc.size()) {
        lfc.read(lbuff)
        rfc.read(rbuff)

        if (not Arrays.equals(lbuff.array(), rbuff.array()))
        {
          return true
        }

        lbuff.clear()
        rbuff.clear()
      }
      return false
    }
  }
```

Gosu ensures that all objects are properly cleaned up. In other words, for each object to create or resource to acquire, if it creates or acquires successfully, Gosu releases, closes, or disposes the object. Also note that if one of the resources fails to create, Gosu does not attempt to acquire other resources in later-appearing items in the command-seperated list. Instead, Gosu simply releases the ones that did succeed.

For more information about using clauses, see “Object Lifecycle Management (‘using’ Clauses)” on page 96 in the *Gosu Reference Guide*.

> **Note:** For more information about concurrency and related APIs in Java, see:
>
> `http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html`

chapter 23

Gosu Programs and Command Line Tools

A *Gosu program* is a file with a `.gsp` file extension that you can run directly from a command-line tool. You can run self-contained Gosu programs outside the ClaimCenter server using the Gosu command line tool. The Gosu shell command-line tool encapsulates the Gosu language engine. You can run Gosu programs directly from the Windows command line as an interactive session or run Gosu program files. You can optionally edit Gosu code with an included lightweight Gosu editor implemented in Java. Using the graphic Gosu editor is a fast way to use the intelligent Gosu code editor and run code without running a full IDE.

Gosu Shell Basics

You can run self-contained Gosu programs outside the ClaimCenter server using the Gosu shell, which is the name for the Gosu language command line tool.

You can use the Gosu shell tool to perform the following tasks:

- Invoke Gosu programs (`.gsp` files). These programs can use other Gosu classes, Gosu extensions, and Java classes.
- Evaluate Gosu expressions interactively using a command-line interface
- Evaluate Gosu expressions passed on the command line

The Gosu shell includes its own lightweight Gosu code editor with intelligent code completion, access to the Gosu type system, and compile time code checking. This version of the code editor simply requires Java, but does not require any other IDEs.

Unpacking and Installing the Gosu Shell

ClaimCenter includes the Gosu shell as a subdirectory called `admin` within the main product installation.

The Gosu shell directory includes the following files and directories:

Path	Purpose
`/bin/gosu.cmd`	The Windows tool that invokes Gosu
`/bin/gosu.sh`	The Unix tool that invokes Gosu (not supported by Guidewire -- not in docs)
`/src/gw/.../*.gs`	Core Gosu classes
`/src/gw/.../*.gsx`	Core Gosu enhancements
`/lib/*.jar`	Java archive (JAR) files that contain core Gosu libraries

To use on another computer, copy the entire Gosu shell directory to a computer with a supported version of the Java run time.

You might consider changing your system's path to add the Gosu shell `bin` directory so you can simply type `gosu` at the command line. On Windows, modify the systemwide `Path` variable by going to the **Start** menu and choosing **Control Panel** → **System** → **Environment Variables**, and choose the `Path` variable. Add a semicolon and the full path to the `bin` directory in the `gosu` shell directory.

For example, suppose you installed (copied) the Gosu shell directory to the path:

```
C:\gosu\
```

Add the following to the system path:

```
;C:\gosu\bin
```

To test this, close any existing command prompt windows, then open a new command prompt window. Type the following command:

```
gosu -help
```

If the help page appears, the Gosu shell is installed correctly.

Gosu Command Line Tool Options

The following table lists the tool command line options:

Task	Options	Example
Run a Gosu program. Include the `.gsp` file extension when specifying the file name.	*`filename`*	`gosu myfile`
Open the graphic Gosu editor, which can also run Gosu code. To open a file, include a filename including the `.gsp` suffix. If you include no filename, the editor creates a new blank unsaved program. Click **Save As** to save the file. **IMPORTANT**: Using the graphic Gosu editor is a fast way to use the intelligent Gosu code editor and run code without running a full IDE.	`-g filename`	`-g myfile`
Evaluate a Gosu expression on command line. Surround the entire expression with quote signs. For any quote sign in the expression, replace it with three double-quote signs. For other special DOS characters such as > and <, precede them with a caret (^) symbol.	-e *expression* -exec *expression*	`gosu -e "new DateTime()"` `gosu -e """"a"""+"""b""""`

Task	Options	Example
Add additional paths to the search path for Java classes or Gosu classes. Separate paths with semicolons. If you are running a `.gsp` file, it is often easier to instead use the `classpath` command within the `.gsp` file rather than this option. For related information, see class loading information in "Calling Classes" on page 322 and "Advanced Class Loading Registry" on page 325.	`-classpath` *path*	`-classpath C:\gosu\projects\libs`
Print help for this tool.	`-h` `-help`	`-h`
Enter interactive shell. Each line you type runs as a Gosu statement. Any results print to the standard output. To exit, type the `exit` or `quit` command. For details, see "Gosu Interactive Shell" on page 325. Also see the standard input option (just a hyphen), discussed later in this table.	`-i` `-interactive`	`-i`
Run a Gosu program entered from the standard input stream. Use this to redirect output of one command as Gosu code into the Gosu shell. For a similar feature, refer to "Gosu Interactive Shell" on page 325	`-`	From DOS command prompt: `echo print(new DateTime()) | gosu -`

Writing a Simple Gosu Program

The following instructions describe running a basic Gosu program after you have installed the shell.

To write and a run a simple Gosu program

1. Open a command prompt.
2. To open the editor, type the command:
   ```
   gosu -g
   ```
3. Type the following line into the editor:
   ```
   print( "Hello World" )
   ```
4. Click **Save As**
5. Navigate to the directory you want to store the Gosu program and save the file as `helloworld.gsp`.
6. Quit the editor using the close box or the **Quit** menu item.
7. In a command prompt window, change the working directory to the directory with the file you created.
8. Type the following command:
   ```
   gosu hello_world.gsp
   ```
   The command line will print:
   ```
   Hello World
   ```

Gosu Program Structure

A simple Gosu program is simple one or more lines of Gosu statements.

Metaline as First Line

Gosu programs support a single line at the beginning of the program for specifying the executable with which to run a file. This is for compliance with the UNIX standard for shell script files. The metaline is optional. If present must the first line of the program. The meta line looks like the following.

```
#!/usr/bin/env gosu
```

Note that the # in the meta line does not mean that the # symbol can start a line comments later on in Gosu programs. The # character is not a valid line comment start symbol.

Functions

Your Gosu program can also define functions in the same file and call them.

For example, the following program creates a simple function and calls it twice:

```
print (sum(10,4,7));
print (sum(222,4,3));


function sum (a: int, b: int, c: int) : int {
  return a + b + c;
 }
```

When run, this program outputs:

```
21
229
```

Calling Classes

You can call out to any Java or Gosu class as needed. However, you cannot define Gosu classes inside your Gosu program file.

To add to the locations that Gosu can load classes, use either the classpath argument on the tool. However, you can also use the `classpath` statement in a Gosu program.

The `classpath` statement in a Gosu program improves upon the normal approach in Java to invoke a full and long classpath as tool option when running the main class.

To set the classpath for a program, simply add `classpath` statements before every other statement in the program. If you use a metaline (see “Metaline as First Line” on page 322), classpath statements appear after the metaline.

A simple version of the classpath statement is simply a relative path in quote signs:

```
classpath "src"
```

If it does not start with a `"/"` character, Gosu treats it **as a relative path**. The path is relative to the folder in which the current program resides. This is the most common use. Use this feature to neatly encapsulate your program and its supporting classes together in one location.

If the path starts with a `"/"` character, Gosu treats it **as an absolute path**.

You can include multiple paths in the same string literal using a comma character as a separator.

For a typical installation, place the Java classes or libraries in a subdirectories of your main Gosu program. For example, suppose you have a Gosu program at this location:

```
C:\gosu\myprograms\test1\test.gsp
```

Copy your class files to locations such as:

```
C:\gosu\myprograms\test1\src\mypackage\myclass.class
```

Copy your library files to locations such as:

```
C:\gosu\myprograms\test1\lib\mylibrary.lib
```

For this example, you would add classpath values with the following statement:

```
classpath "src,lib"
```

Command Line Arguments

There are two ways you can access command line arguments to programs:

- **Manipulating raw arguments.** You can get the full list of arguments as they appear on the command line. If any option has multiple parts separated by space characters (such as `"-username jsmith"`), each component is a separate raw argument.
- **Advanced argument processing.** You can use parse the command line for options with a hyphen prefix and optional additional values associated with the preceding command line option. For example, `"-username jsmith"` is a single option to set the `username` option to the value `jsmith`.

Raw Argument Processing

To get the full list of command line arguments as a list of `String` values, use the `CommandLineAccess` class. Call its `getRawArgs` method, which returns an array of String values.

```
uses gw.lang.cli.CommandLineAccess
print( "CommandLineArgs: " + CommandLineAccess.getRawArgs() )
```

Advanced Argument Processing

A more advanced way to access command line arguments is to write your own class that populate all your properties from the individual command line options. This approach supports Boolean flags or setting values from the command line.

This approach requires you to define a simple Gosu class upon which you define static properties. Define one static property for each command line option. Static properties are properties stored exactly once on the class itself, rather than on instances of the class.

You can then initialize those properties by passing your custom class to the `CommandLineAccess.initialize(...)` method. The `initialize` method overrides the static property values with values extracted from the command line. After processing, you can use an intuitive Gosu property syntax to get the values from the static properties in your own Gosu class.

First, create a Gosu class that defines your properties. It does not need to extend from any particular class. The following example defines two properties, one `String` property named `Name` and an a `boolean` property called `Hidden`:

```
package test
uses gw.lang.cli.*

class Args {
  // String argument
  static var _name : String as Name

  // boolean argument -- no value to set on the command line
  static var _hidden : boolean as Hidden
}
```

Note that the publicly-exposed property name is the symbol after the "`as`" keyword (in this case `Name` and `Hidden`), not the private static variable itself. These are the names that are the options, although the case can vary, such as: "`-name jsmith`" instead of "`-Name jsmith`".

Choose a directory to save your command line tool. Create a subdirectory named `src`. Inside that create a subdirectory called `test` (the package name). Save this Gosu class file as the file `Args.gs` in that `src/test` directory.

Next, run the following command

Paste in the following code for your program:

```
classpath "src"
```

```
uses gw.lang.cli.*
uses test.*

CommandLineAccess.initialize ( Args)

print("hello " + Args.Name)
print("you are " + (Args.Hidden ? "hidden" : "visible") + "!!!!")
```

Click **Save As** and save this new command line tool as `myaction.gsp` in the directory two levels up from the `Args.gs` file.

From the command batch window, enter the following command

```
gosu myaction.gsp -name John -hidden false
```

This outputs:

```
hello John
you are visible!!!!
```

One nice benefit of this approach is that these properties are available globally to all Gosu code as static properties. After initialization, all Gosu code can access properties merely by accessing the type (the class), without pass a object instance to contain the properties.

Note that you can access the properties uncapitalized to better fit normal command line conventions.

The `String` property we define requires an argument value to follow the option. This is true of all non-boolean property types. However, the boolean property does not require an argument value, and this type is special for this reason. If a property is defined to have type `boolean` and the option is specified with no following value, Gosu assumes the value `true` by default.

The properties can be any type to work with this approach, not merely `String` and `boolean`. However, there must exist a Gosu coercion of the type from a `String` in order to avoid exceptions at run time. If no coercion exists, a workaround is to add a writable property of type `String`,and add a read-only property that transforms that `String` appropriately. This read-only property allows you to do whatever deserialization logic you would like, all defined in Gosu.

Only properties defined with the modifiers `public`, `static`, `writable` properties on your command line class participate in command line argument initialization.

If a user enters an incorrect option, `CommandLineAccess.initialize()` prints a help message and exits with a -1 return code. If you do not want this exit behavior, there is a secondary (overloaded) version of the `initialize` method that you can use instead. Simply add the value `false` a second parameter to the method to suppress exiting on bad arguments.

Special Annotations for Command Line Options

You can use Gosu annotations from the `gw.lang.cli.*` package on the static properties defined in your command line class. Simply add one of the following annotation lines immediately before the line that defines the property:

Annotation	Description
`@Required()`	This command line tool will not parse unless this property is included
`@DefaultValue( String )`	The default string value of this property.
`@ShortName( String )`	The short name of this option when used with a single-dash argument. For example, a property named `Day` preceded by the annotation `@ShortName( "d" )` allows the option `"-d"` shortcut instead of “`-day`”. The short name works with the single-dash argument but not the double-dash variant.

For example:

```
package test
uses gw.lang.cli.*

class Args {
```

```
// String argument
@Required()
static var _name : String as Name

// boolean argument -- no value to set on the command line
@ShortName("s")
static var _hidden : boolean as Hidden
}
```

Advanced Class Loading Registry

An alternative to using the `classpath` directive directly in the program is to use a `registry.xml` file in the same directory as your program file. The registry file gives you additional control over your Gosu environment, including the ability to specify additional type loaders. It is also useful when you have a lot of programs that share the same configuration environment.

The structure of the `registry.xml` file is as following:

```
<?xml version="1.0" encoding="UTF-8"?>

<serialization xmlns="http://guidewire.com/xml" xmlns:tns="http://guidewire.com/xml"
xmlns:xsd="http://www.w3.org/1999/XMLSchema" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">

  <common-service-init class="gw.internal.gosu.ShellKernelInit"/>

  <classpath>
    <entry>../lib</entry>
    <entry>../gsrc</entry>
  </classpath>

</serialization>
```

To add paths to the class path, add more `<entry>` elements containing class paths.

The Self-Contained Gosu Editor

Gosu ships with a simple editor application for writing small Gosu programs. If you run the `gosu` command line tool without any arguments and your run environment supports Java windows programs, the Gosu editor runs.

You can also launch the editor and preload with a specific program using the `-t` flag:

```
C:\eng\pl\carbon\active\gosu>gosu-dist\bin\gosu -t Foo.gsp
```

Gosu Interactive Shell

Gosu includes an interactive text-based shell mode. Each line you type runs as a Gosu statement, and any results print to the standard output.

To enter the interactive shell, run the Gosu batch file in the bin directory with the `-i` option:

```
gosu -i
```

Or, simply run the tool from the command with no extra options to enter interactive mode:

```
gosu
```

The program will display a prompt that indicates that you are in the interactive shell rather than the command prompt environment that called this tool.

```
gs >
```

You can then enter a series of Gosu expressions and statements, including defining functions and variables.

For example, you can type the following series of lines at the prompt:

```
gs > var s = new java.util.ArrayList() {1,2,3}
gs > s.each( \ o : Object -> print(o))
```

The Gosu shell will output the following:

```
1
2
3
```

Type the command `help` to see all available commands in the interactive shell. Additional commands in the interactive shell include the following:

Command	Description
`quit`	Quit the interactive shell.
`exit`	Quit the interactive shell
`ls`	Show a list of all defined variables
`rm` *VARNAME*	remove a variable from interactive shell memory
`clear`	clears (removes) all variables from interactive shell memory

If you enter a line of Gosu that necessarily requires additional lines, Gosu displays a different prompt (“`...`”) for you to type the remaining lines. For example, if you type a statement block with an opening brace but no closing brace, you can enter the remaining lines in the statement block. After you enter the line with the closing brace, the shell returns to its regular prompt.

The shell provides code-completion using the **TAB** key. You must type at least one letter of a symbol, after which you can type TAB and the shell will display various options. Note that package completion is not supported.

For example, type the following lines but do not press enter on the last line yet:

```
gs > var s = new java.util.ArrayList() {1,2,3}
gs > s.e
```

If you press TAB, the shell displays properties and methods that begin with the letter “e” then redisplays the current line you are typing:

```
each( block( java.lang.Object ):void )     eachWithIndex( block( java.lang.Object, int ):void )
elementAt( int )                           ensureCapacity( int )
equals( java.lang.Object )                 except( java.lang.Iterable<java.lang.Object> )

gs > s.e
```

To exit, type the `exit` or `quit` command.

Notes:

- Functions and blocks are supported in the interactive shell. However, defining new Gosu classes is not supported in the interactive shell.
- The interactive shell is different from the *standard in* option for the tool, which may be appropriate for some purposes. You can define the output of one tool to be in Gosu and then redirect (*pipe*) the contents of that tool into the Gosu shell, using the hyphen option.

Helpful APIs for Command Line Gosu Programs

Read Line

Use the `readLine` API to reads a line of input from the console using the given prompt. For example:

```
var res = gw.util.Shell.readLine("Are you sure you want to delete that directory?")
```

Is Windows

Call the `gw.util.Shell.isWindows()` method to determine if the current host system is Windows-based.

chapter 24

Running Local Shell Commands

You can run local command line programs from Gosu.

Running Command Line Tools from Gosu

You can run local command line programs from Gosu. These APIs execute the given command as if it had been executed from the command line of the host operating system.

> **Note:** This feature is separate from the feature to run Gosu as a self-contained language outside ClaimCenter.

The Gosu class `gw.util.Shell` provides methods to run local command-line programs. For example, it can run `cmd.exe` scripts on Windows or `/bin/sh` on Unix. Gosu returns all content that is sent to standard out as a Gosu String. If the command finishes with a non zero return value, Gosu throws a `CommandFailedException` exception.

Content sent to standard error is forwarded to standard error for this Java Virtual Machine (JVM). If you wish to capture `StdErr` as well, use the `buildProcess(String)` method to create a `ProcessStarter` and call the `ProcessStarter.withStdErrHandler(gw.util.ProcessStarter.OutputHandler)` method.

> **IMPORTANT** This method blocks on the execution of the command.

Pass the command as a `String` to the `exec` method.

For example:

```
var currentDir = Shell.exec( "dir" ) // windows
var currentDir = Shell.exec( "ls" )  // *nix
Shell.exec( "rm -rf " + directoryToDelete ) // directory remove on Unix
```

On windows, Gosu uses `CMD.EXE` to interpret commands. Beware of problems due to limitations of `CMD.EXE`, such as a command string may be too long for it. In these cases consider the `buildProcess(String)` method instead.

For related tools, see “Helpful APIs for Command Line Gosu Programs” on page 326 in the *Gosu Reference Guide*.

chapter 25

Checksums

This topic describes APIs for generating *checksums*. Longer checksums such as 64-bit checks sums are also known as *fingerprints*. Send these fingerprints along with data to improve detection from accidental modification of data in transit. For example, detecting corrupted stored data or errors in a communication channel.

This topic includes:

Overview of Checksums

To improve detection of accidental modification of data in transit, you can use *checksums*. A checksum is a computed value generated from an arbitrary block of digital source data. To check the integrity of the data at a later time, recompute the checksum and compare it with the stored checksum. If the checksums do not match, the data was almost certainly altered (either intentionally or unintentionally). For example, this technique can help detection of physical data corruption or errors in a communication channel.

Be aware that checksums cannot perfectly protect against intentional corruption by a malicious agent. A malicious attacker could modify the data so as to preserve its checksum value, or depending on the transport could substitute a new checksum for the modified data. To guard against malicious changes, use encryption at the data level (a cryptographic hash) or the transport level (such as SSL/HTTPS).

WARNING Checksums improve detection from accidental modification of data but cannot detect intentional corruption by a malicious agent. If you need that level of protection, use encryption instead of checksums, or in addition to checksums.

You can also use fingerprints to design caching and syncing algorithms that check whether data changed since the last cached copy. You can save the fingerprint of the cached copy and an external system can generate a fingerprint of its most current data. If you have both fingerprints, compare them to determine if you must resync the data. To work effectively, the fingerprint algorithm must provide near-certainty that a real-world change would change the fingerprint. In essence, a fingerprint uniquely identifies the data for most practical purposes, although in fact collisions (changed data with matching fingerprints) is theoretically possible.

Guidewire provides support for 64-bit checksums in the class `FP64` in the package `gw.util.fingerprint`.

The `FP64` class provides methods for computing 64-bit fingerprints of the following kinds of data:

- `String` objects
- character arrays
- byte arrays
- input streams

This implementation is based on an original idea of Michael O. Rabin, with refinements by Andrei Broder. Fingerprints provide a probabilistic guarantee that defines a mathematical upper bound on the probability of a collision (a collision occurs if two different strings have the same fingerprint). Using 64-bit fingerprints, the odds of a collision are extremely small. The odds of a collision between two randomly chosen texts a million characters long are less than 1 in a trillion.

Suppose you have a set `S` of `n` distinct strings each of which is at most `m` characters long. The odds of any two different strings in `S` having the same fingerprint is described by the following equation (`k` is the number of bits in the fingerprint):

```
(nm^2) / 2^k
```

For practical purposes, you can treat fingerprints as uniquely identifying the bytes that produced them. In mathematical notation given two `String` variables `s1` and `s2`,using the → symbol to mean "*implies*":

```
new FP64(s1).equals(new FP64(s2)) → s1.equals(s2)
```

Do not fingerprint the value of (the raw bytes of) a fingerprint. In other words, do not fingerprint the output of the `FP64` methods `toBytes` and `toHexString`. If you do so, due to the shorter length of the fingerprint itself, the probabilistic guarantee is invalid and may lead to unexpected collisions.

Creating Fingerprints

To create a fingerprint object, use the constructor to the FP64 object and pass it one of the supported objects:

An example of passing a `String` object:

```
var s = "hello"
var f = new FP64(s)
```

An example of passing a character array:

```
var s = "hello"
var ca : char[] = {s[0], s[1], s[2], s[3], s[4]}
var f = new FP64(ca)
```

Note: There is an alternate method signature that takes extra parameters for start position and length of the desired series of characters in the array.

An example of passing a byte array:

```
var ba = "hello".Bytes // or use "hello".getBytes(
var f = new FP64(ba)
```

Note: There is an alternate method signature that takes extra parameters for start position and length of the desired series of byes in the array.

An example of passing a stream:

```
var s = "testInputStreamConstructor"
new FP64(new ByteArrayInputStream(gw.util.StreamUtil.toBytes(s))));
```

An example of passing an input stream:

```
var s = "testInputStreamConstructor"
new FP64(new StringBuffer(g));
```

An example of passing another `FP64` fingerprint object to the constructor to duplicate the fingerprint:

```
var s = "hello"
var f = new FP64(s)
var f2 = new FP64(f)
```

How to Output Data Inside a Fingerprint

To generate output data from a finger print, use the `FP64` method `toBytes()`, which returns the value of this fingerprint as a newly-allocated array of 8 bytes.

Instead of the no-argument method, you can also use the alternate method signature that takes a byte array buffer and the method writes the bytes there. The buffer must have length at least 8 bytes.

Alternatively, you can use a method `toHexString()`. This method returns the fingerprint as an unsigned integer encoded in base 16 (hexadecimal) and padded with leading zeros to a total length of 16 characters.

Extending Fingerprints

This class also provides methods for *extending* an existing fingerprint by more bytes or characters. This is useful if you are sure the only change to the source data was appending a known series of bytes to the **end** of the original `String` data.

Extending the fingerprint of one `String` by another `String` produces a fingerprint equivalent to the fingerprint of the concatenation of the two String objects. Given the two `String` variables `s1` and `s2`, this means the following is true:

```
new FP64(s1 + s2).equals( new FP64(s1).extend(s2) )
```

All operations for extending a fingerprint are **destructive**. In other words, they modify the fingerprint object directly (*in-place*). All operations return the resulting `FP64` object, so you can chain method calls together, such as the following:

```
new FP64("x").extend(foo).extend(92))
```

If you want to make a copy of a fingerprint, use the FP64 constructor and pass the FP64 object to copy:

```
var original = new FP64("Hello world")
var copy = new FP64(original) // a duplicate of the original fingerprint
```

chapter 26

Coding Style

This topic lists some Gosu coding practices that Guidewire recommends. These guidelines encourage good programming practices that improve Gosu readability and encourage code that is error-free, easy to understand, and easy to maintain by other people.

This topic includes:

General Coding Guidelines

Omit Semicolons

Omit semicolons, as they are unnecessary in almost all cases. Gosu code looks cleaner this way.

Semicolons are only needed if separating multiple Gosu statements all written on one line within a one-line statement list. Even this is generally not recommended, although it is sometimes appropriate for simple statement lists declared in-line within Gosu block definitions.

Type Declarations

Omit the type declaration if you declare variables with an assignment. Instead, use “`as` *`TYPE`*” where appropriate. The type declaration is particularly redundant if a value needs coerce to a type already included at the end of the Gosu statement.

In other words, the recommended type declaration style is:

```
var delplans = currentPage as DelinquencyPlans
```

Do **not** add the redundant type declaration:

```
var delplans : DelinquencyPlans = currentPage as DelinquencyPlans
```

The == and != Operator Recommendations and Warnings

The Gosu `==` and `!=` operators are safe to use even if one side evaluates to `null`.

Use these operators where possible instead of using the `equals` method on objects. This protection with `null` is called *null-safety*.

Notice that Gosu's == operator is equivalent to the object method `equals` (`obj1.equals(obj2)`) other than its difference in null-safety.

> **Note:** For those who use the Java language also, the null-safety of the Gosu == operator is similar to the null-safety of the Java code `ObjectUtil.equals(...)`. In contrast, for both the Gosu and Java languages, the object method *`myobject`*`.equals(...)` is not null-safe.

So, any Gosu code that use the `equals` method, such as:

```
( planName.equals( row.Name.text ) )
```

...can be written in easier-to-understand code as:

```
( planName == row.Name.text) )
```

Although the == and the `!=` comparison operators are more powerful and more convenient than `equals()`, be aware of coercions that may occur. For example, because expressions adhere to Gosu's implicit coercion rules, the expression `1 == "1"` evaluates to `true`. In other words, the number 1 and the string representing the number 1 is true. This is because of implicit coercion that allows the string `"1"` to be assigned to an integer variable as the integer 1 without explicit casting.

While coercion behavior is convenient and powerful, it can be dangerous if used carelessly. Gosu produces compile warnings for implicit coercions. Take the warnings seriously and in most cases *explicitly* cast using the `as` keyword in cases that you want the coercion. Otherwise, fix the problem by rewriting in some other way entirely.

For example, an expression equates a date value with a string representation of a date value:

```
(dateVal == strVal)
```

It is safest to rewrite this as the following:

```
(dateVal == strVal as DateTime)
```

Carefully consider any implicit direct coercions that might occur with the == operator, and explicitly define coercions where possible.

If comparing array equality with the == and `!=` operators, Gosu does not let you compare *incompatible* array types. For example, the following code generates a compile time error because arrays of numbers and strings are incompatible:

```
new Number[] {1,2} == new String[] {"1","2"}
```

However, if the array types are comparable, Gosu recursively applies implicit coercion rules on the array's **elements**. For example, the following code evaluates to `true` because a `Number` is a subclass of `Object`, so the Gosu compares the individual elements of the table:

```
new Number[] {1,2} == new Object[] {"1","2"}
```

> **WARNING** Be careful if comparing arrays. Note the recursive comparison of individual elements for compatible array types.

For more information about the difference between == and === operators in Gosu, see "=== Operator Compares Object Equality" on page 66

Gosu Case Sensitivity Implications

Gosu compiles and run faster if you write all your Gosu as case-sensitive code. Guidewire strongly recommends that you always use the proper capitalization precisely as defined for all of the following

- proper type names
- variable names
- keywords (such as `var` and `if`)

- method names
- property names
- package names
- all other language elements.

In addition to speed benefits, using proper capitalization makes your code easier to read.

For more examples and additional information on this topic, see "Gosu Case Sensitivity" on page 31.

Class Variable and Class Property Recommendations

Always prefix `private` and `protected` class variables with an underscore character (_).

Avoid *public variables*. Convert public variables to properties, so that the interface to other code (the property names) is separated from the implementation (the storage and retrieval).

Although public variables are a supported part of the language for compatibility with other languages, Guidewire strongly recommends public properties backed by private variables instead of using public variables. You can do this easily in Gosu on the same line as the variable definition using the **as** keyword.

In other words, in your new Gosu classes that define class variables, do this:

```
private var _firstName : String as FirstName
```

Do not do this:

```
public var FirstName : String
```

For more information about defining properties, see "Properties" on page 172.

IMPORTANT For Gosu classes data fields, Guidewire strongly recommends public properties backed by private variables rather than public variables. Do not use public variables in new Gosu classes. See "Properties" on page 172 for more information.

Use 'typeis' Inference

To improve the readability of your Gosu code, Gosu automatically downcasts after a `typeis` expression if the type is a subtype of the original type. This is particularly valuable for `if` statements and similar Gosu structures. Within the Gosu code bounded by the `if` statement, you do not need to do casting ("`as` *`TYPE`*" expressions) to that subtype. Because Gosu confirms that the object has the more specific subtype, Gosu implicitly considers that variable's type to be the **subtype**, at least within that block of code.

The structure of this type looks like the following:

```
var VARIABLE_NAME : TYPE_NAME

if (VARIABLE_NAME typeis SUBTYPE_NAME) {

  // use the VARIABLE_NAME as SUBTYPE_NAME without casting
  // This assumes SUBTYPE_NAME is a subtype of TYPE_NAME
}
```

For example, the following example shows a variable declared as an `Object`, but downcasted to `String` within the `if` statement in a block of code within an `if` statement.

Because of downcasting, the following code is valid:

```
var x : Object = "nice"
var strlen = 0

if( x typeis String ) {
  strlen = x.length
}
```

This works because the `typeis` inference is effective immediately and propagates to adjacent expressions.

It is important to note that `length` is a property on `String`, not `Object`. The downcasting from `Object` to `String` means that you do not need an additional casting around the variable `x`. In other words, the following code is equivalent but has an **unnecessary** cast:

```
var x : Object = "nice"
var strlen = 0

if( x typeis String ) {
  strlen = (x as String).length // "length" is a property on String, not Object
}
```

Use automatic downcasting to write easy-to-read and concise Gosu code. Do not write Gosu code with unnecessary casts. For more information, see "Automatic Downcasting for 'typeis' and 'typeof'" on page 302.