



Gosu Reference Guide

ClaimCenter RELEASE 8.0.2



Copyright © 2001-2014 Guidewire Software, Inc. All rights reserved.

Guidewire, Guidewire Software, Guidewire ClaimCenter, Guidewire PolicyCenter, Guidewire BillingCenter, Guidewire Reinsurance Management, Guidewire ContactManager, Guidewire Vendor Data Management, Guidewire Client Data Management, Guidewire Rating Management, Guidewire InsuranceSuite, Guidewire ContactCenter, Guidewire Studio, Guidewire Product Designer, Guidewire Live, Guidewire DataHub, Guidewire InfoCenter, Guidewire Standard Reporting, Guidewire ExampleCenter, Guidewire Account Manager Portal, Guidewire Claim Portal, Guidewire Policyholder Portal, ClaimCenter, BillingCenter, PolicyCenter, InsuranceSuite, Gosu, Deliver Insurance Your Way, and the Guidewire logo are trademarks, service marks, or registered trademarks of Guidewire Software, Inc. in the United States and/or other countries.

All other trademarks are the property of their respective owners.

This material is confidential and proprietary to Guidewire and subject to the confidentiality terms in the applicable license agreement and/or separate nondisclosure agreement.

Guidewire products are protected by one or more United States patents.

Product Name: Guidewire ClaimCenter

Product Release: 8.0.2

Document Name: Gosu Reference Guide

Document Revision: 20-May-2014



Contents

About ClaimCenter Documentation	13
Conventions in This Document	14
Support	14
1 Gosu Introduction	15
Welcome to Gosu	15
Control Flow	17
Blocks	18
Enhancements	19
Collections	19
Access to Java Types	20
Gosu Classes and Properties	20
Interfaces	24
List and Array Expansion Operator *	24
Comparisons	24
Case Sensitivity	25
Compound Assignment Statements	25
Delegating Interface Implementation with Composition	26
Concurrency	26
Exceptions	27
Annotations	28
Gosu Templates	28
XML and XSD Support	28
Web Service Support (Consuming WSDL)	29
Gosu Character Set	30
Running Gosu Programs and Calling Other Classes	30
More About the Gosu Type System	31
Compile Time Error Prevention	31
Type Inference	32
Intelligent Code Completion and Other Gosu Editor Tools	32
Null Safety for Properties and Other Operators	32
Generics in Gosu	34
Gosu Primitives Types	35
Gosu Case Sensitivity and Capitalization	35
Gosu Statement Terminators	35
Gosu Comments	36
Gosu Reserved Words	36
Gosu Generated Documentation ('gosudoc')	37
Code Coverage Support	37
Notable Differences Between Gosu and Java	38
Get Ready for Gosu	42
2 Types	43
Access to Java Types	43
Primitive Types	44

Objects	44
Object Instantiation.....	45
Object Property Assignment.....	45
Object Property Access.....	46
Object Methods.....	47
Boolean Values.....	47
Sequences of Characters.....	48
Array Types	52
List Access Using Array Index Notation	52
Array Expansion	53
Array-related Entity Methods	53
Associative Array Syntax for Property Access.....	54
Legacy Array Type	55
Numeric Literals	55
Entity Types	56
Typekeys and Typelists	56
Typelist Literals	56
Typekey Literals	56
Getting Information from a Typekey	58
Compatibility with Earlier Gosu Releases	58
DateTime.....	59
Number	60
Array	61
3 Gosu Operators and Expressions.....	63
Gosu Operators	63
Operator Precedence.....	64
Standard Gosu Expressions	65
Arithmetic Expressions.....	65
Equality Expressions.....	68
Evaluation Expressions.....	70
Existence Testing Expressions	70
Logical Expressions	71
New Object Expressions.....	73
Relational Expressions	76
Unary Expressions	78
Importing Types and Package Namespaces	79
Conditional Ternary Expressions	80
Special Gosu Expressions	82
Function Calls	82
Static Method Calls.....	82
Static Property Paths.....	82
Entity and Typekey Type Literals.....	83
Handling Null Values In Expressions.....	83
Null-safe Property Access.....	83
Null-safe Default Operator	85
Null-safe Indexing for Arrays, Lists, and Maps	85
Null-safe Math Operators	85
4 Statements.....	87
Gosu Statements	87
Statement Lists	87
New Is Optionally a Statement	88

Gosu Variables	88
Variable Type Declaration	88
Variable Assignment.....	89
Gosu Conditional Execution and Looping	92
If - Else Statements	92
For Statements.....	93
While() Statements	94
Do...While() Statements.....	95
Switch() Statements	95
Gosu Functions	96
Named Arguments and Argument Defaults	98
Public and Private Functions.....	99
5 Exception Handling.....	101
Try-Catch-Finally Constructions	101
Throw Statements	102
Catching Exceptions in Gosu	103
Object Lifecycle Management (using Clauses)	104
Disposable Objects	105
Closeable Objects and using Clauses	106
Reentrant Objects and using Clauses	107
Returning Values from using Clauses.....	108
Optional Use of a finally Clause with a using Clause.....	109
Assert Statements	110
6 Intervals.....	111
What are Intervals?.....	111
Reversing Interval Order.....	112
Granularity (Step and Unit).....	113
Writing Your Own Interval Type	113
Custom Iterable Intervals Using Sequenceable Items.....	113
Custom Iterable Intervals Using Manually-written Iterators	115
Custom Non-iterable Interval Types.....	118
7 Calling Java from Gosu	119
Overview of Writing Gosu Code that Calls Java	119
Many Java Classes are Core Classes for Gosu	120
Java Packages in Scope	120
Static Members and Static Import in Gosu	120
Java get/set/is Methods Convert to Gosu Properties.....	121
Interfaces	123
Enumerations	123
Annotations	123
Java Primitives	123
Java Generics	124
8 Query Builder APIs	125
Overview of the Query Builder APIs	125
The Processing Cycle of Queries	125
SQL Select Statements and Query Builder APIs Compared	127
Building Simple Queries.....	128
Restricting the Results of a Simple Query	129
Ordering the Results of a Simple Query	129
Accessing the Results of a Simple Query	129

Joining Related Entities to Queries	130
Joining an Entity to a Query with a Simple Join	131
Restricting Query Results with Fields on Joined Entities	132
Different Ways to Join Related Entities to Queries	133
Making a Query with an Inner Join	133
Making a Query with a Left Outer Join	136
Adding Predicates to Joined Entities	137
Handling Duplicates in Joins with Foreign Keys on the Right	138
Restricting Queries with Predicates on Fields	139
Using Comparison Predicates with Character Fields	140
Using Comparison Predicates with Date and Time Fields	141
Using Comparison Predicates with Null Values	143
Using Set Inclusion and Exclusion Predicates	144
Comparing Column Values with Each Other	145
Comparing Column Values with Literal Values	145
Comparing Typekey Column Values with Typekey Literals	146
Combining Predicates with AND and OR Logic	147
Predicate Methods Reference	151
Working with Row Queries	153
Setting Up Row Queries	153
Database Aggregate Functions Within Select Blocks	155
Applying Functions to Selected Fields	156
Limitations of Row Queries	158
Working with Results	158
What Result Objects Contain	158
Filtering Results with Standard Query Filters	160
Ordering Results	165
Useful Properties and Methods on Result Objects	166
Converting Result Objects to Lists, Arrays, Collections, and Sets	169
Updating Entity Instances in Query Results	170
Testing and Optimizing Queries	171
Performance Differences Between Entity and Row Queries	171
Viewing the SQL Select Statement for a Query	172
Enabling Context Comments in Queries on SQL Server or DB2	173
Including Retired Entities in Query Results	174
Setting the Page Size for Prefetching Query Results	174
Chaining Query Builder Methods	174
Working with Advanced Inline Views	175
Method and Type Reference for the Query Builder APIs	176
9 Find Expressions	179
Basic Find Expressions	179
Find Expressions that Use AND/OR Operators	180
Find Expressions that Use Equality and Relational Operators	181
Find Expressions that Use Where...In Clauses	182
Find Expressions and Non-Equality Comparisons	182
Using Exists Expressions for Array Properties in Find Expressions	183
Fixing Invalid Queries by Adding Exists Clauses	184
Combining Exists Expressions	184
Find Expressions that Use Special Substring Keywords	184

Using the Results of Find Expressions (Using Query Objects)	185
Basic Iterator Example	185
Handling Large Query Objects from Find Expressions	185
Sorting Results	186
Retrieving a Single Row from Find Expression Results.....	186
Found Entities Are Read-only Until Added to a Bundle	186
Query Objects Returned by Find Expressions Are Always Dynamic	187
10 Classes.....	189
What Are Classes?	189
Creating and Instantiating Classes	190
Creating a New Instance of a Class.....	192
Naming Conventions for Packages and Classes.....	192
Properties	193
Properties Act Like Data But They Are Dynamic and Virtual Functions	194
Property Paths are Null Tolerant.....	194
Static Properties	197
More Property Examples.....	197
Modifiers.....	198
Access Modifiers.....	199
Override Modifier.....	200
Abstract Modifier	200
Final Modifier	201
Static Modifier	204
Inner Classes.....	205
Named Inner Classes.....	205
Anonymous Inner Classes.....	206
11 Enumerations	209
Using Enumerations	209
Extracting Information from Enumerations	210
Comparing Enumerations	210
12 Interfaces.....	211
What is an Interface?	211
Defining and Using an Interface.....	212
Defining and Using Properties with Interfaces	213
Modifiers and Interfaces	214
13 Composition	215
Using Gosu Composition	215
Overriding Methods Independent of the Delegate Class	217
Declaring Delegate Implementation Type in the Variable Definition	217
Using One Delegate for Multiple Interfaces.....	218
Using Composition With Built-in Interfaces	218
14 Annotations.....	219
Annotating a Class, Method, Type, Class Variable, or Argument	219
Built-in Annotations	220
Annotations at Run Time	222
Gosu Class Function Parameter Argument Annotations at Run Time	223
Defining Your Own Annotations	223
Customizing Annotation Usage	225

15 Enhancements	227
Using Enhancements	227
Syntax for Using Enhancements	228
Creating a New Enhancement	228
Syntax for Defining Enhancements	228
Enhancement Naming and Package Conventions	230
Enhancements on Arrays	230
16 Gosu Blocks	231
What Are Blocks?	231
Basic Block Definition and Invocation	232
Variable Scope and Capturing Variables In Blocks	234
Argument Type Inference Shortcut In Certain Cases	235
Block Type Literals	235
Blocks and Collections	237
Blocks as Shortcuts for Anonymous Classes	237
17 Gosu Generics	239
Gosu Generics Overview	240
Using Gosu Generics	241
Parameterized Classes	242
Parameterized Methods	243
Other Unbounded Generics Wildcards	243
Generics and Blocks	244
How Generics Help Define Collection APIs	246
Multiple Dimensionality Generics	247
Generics With Custom ‘Containers’	247
Generics with Non-Containers	248
18 Collections	251
Basic Lists	251
Creating a List	251
Type Inference and List Initialization	252
Getting and Setting List Values	252
Special Behavior of List Interface in Gosu	253
Basic Hash Maps	253
Creating a Hash Map	253
Getting and Setting Values in a Hash Map	253
Creating a Hash Map with Type Inference	253
Special Enhancements on Maps	254
Wrapped Maps with Default Values	254
List and Array Expansion (*.)	255
Array Flattening to Single Dimensional Array	256
Application-Specific Examples	257
Enhancement Reference for Collections and Related Types	257
Collections Enhancement Methods	257
Finding Data in Collections	261
Sorting Collections	261
Mapping Data in Collections	262
Iterating Across Collections	263
Partitioning Collections	263
Converting Lists, Arrays, and Sets	264
Flat Mapping a Series of Collections or Arrays	264
Sizes and Length of Collections and Strings are Equivalent	265

19 Gosu and XML.....	267
Manipulating XML Overview	268
Legacy XML Support	268
Introduction to the XML Element in Gosu.....	271
Dollar Sign Prefix for Properties that Are XSD Types.....	274
Exporting XML Data	275
Export-related Methods on an XML Element.....	275
XML Serialization Options Reference and Examples	276
Parsing XML Data into an XML Element	278
Creating Many Qualified Names in the Same Namespace.....	280
XSD-based Properties and Types	281
Important Concepts in XSD Properties and Types	281
XSD Generated Type Examples	285
Automatic Insertion into Lists.....	286
XSD List Property Example	287
Getting Data From an XML Element	288
Manipulating Elements and Values (Works With or Without XSD)	288
Attributes.....	291
Simple Values.....	291
Methods to Create XML Simple Values.....	292
XSD to Gosu Simple Type Mappings.....	293
Facet Validation	293
Access the Nullness of an Element	294
Automatic Creation of Intermediary Elements.....	295
Default and Fixed Attribute Values	295
Substitution Group Hierarchies	296
Element Sorting for XSD-based Elements	297
Built-in Schemas.....	300
The XSD that Defines an XSD (The Metaschema)	300
Use a Local XSD for an External Namespace or XSD Location.....	301
Schema Access Type	301
The Guidewire XML (GX) Modeler	302
Automatic Publishing of the Generated XSD	307
Generating XML Using an XML Model	307
Customizing GX Modeler Output (GXOptions)	308
Parsing XML Into an XML Model	311
Arrays of Entities in XML Output	311
Complete Guidewire XML Model Example.....	312
XML Serialization Performance and Element Sorting	313
Type Conversions from Gosu Types to XSD Types.....	313
Legacy XML APIs: Manipulating XML as Untyped Nodes	313
Untyped Node Operations.....	314
Example of Manipulating XML as Untyped Nodes	315
Legacy XML APIs: Exporting XML Data.....	316
Legacy XML APIs: Collection-like Enhancements for XML	318
Legacy XML APIs: Structured XML Using XSDs	319
Importing Strongly-Typed XML	322
Writing Strongly-Typed XML	323
Handling XSD Choices in XML.....	324
Gosu Type to XSD Type Conversion Reference	326
XSD Namespaces and Qualified Names.....	328
Autocreation of Intermediate Nodes	329
XML Node IDs	329
Date Handling in XSDs.....	330

20 Bundles and Database Transactions	331
When to Use Database Transaction APIs.....	332
Bundle Overview	334
Adding Entity Instances to Bundles	334
Making an Entity Instance Writable By Adding to a Bundle.....	334
Moving a Writable Entity Instance to a New Writable Bundle	335
Getting the Bundle of an Existing Entity Instance	335
Getting an Entity from a Public ID or a Key (Internal ID).....	336
Creating New Entity Instances in Specific Bundles.....	336
Committing a Bundle Explicitly in Very Rare Cases.....	337
Removing Entity Instances from the Database.....	337
Determining What Data Changed in a Bundle	338
Detecting Property Changes on an Entity Instance.....	340
Getting Changes to Entity Arrays in the Current Bundle	340
Getting Add, Changed, or Deleted Entities In a Bundle.....	341
Running Code in an Entirely New Bundle	342
Create Bundle For a Specific ClaimCenter User	344
Warning about Transaction Class Confusion	344
Exception Handling And Database Commits	344
Bundles and Published Web Services.....	344
Entity Cache Versioning, Locking, and Entity Refreshing	344
Entity Instance Versioning and the Entity Touch API	345
Record Locking for Concurrent Data Access	345
User Interface Bundle Refreshes.....	345
Details of Bundle Commit Steps	346
21 Gosu Templates	347
Template Overview.....	347
Template Expressions	347
When to Escape Special Characters for Templates.....	348
Using Template Files	349
Creating and Running a Template File.....	350
Template Scriptlet Tags	350
Template Parameters.....	351
Extending a Template From a Class	352
Template Comments	353
Template Export Formats	353
22 Type System	355
The Type of All Types	355
Basic Type Coercion.....	356
Basic Type Checking	357
Automatic Downcasting for ‘typeis’ and ‘typeof’	358
Using Reflection	360
Type Object Properties	362
Java Type Reflection.....	364
Type System Class	364
Feature Literals	365
Compound Types	367
Type Loaders	367
23 Concurrency	369
Overview of Thread Safety and Concurrency	369
Request and Session Scoped Variables	371

Concurrent Lazy Variables	372
Optional Non-Locking Lazy Variables	373
Concurrent Cache	373
Concurrency with Monitor Locks and Reentrant Objects	374
24 Gosu Command Line Shell	377
Gosu Command Line Tool Basics	377
Unpacking and Installing the Gosu Command Line Shell	377
Command Line Tool Options	378
Writing a Simple Gosu Command Line Program	379
Command Line Arguments	379
Advanced Class Loading Registry	381
Gosu Interactive Shell	382
Helpful APIs for Command Line Gosu Programs	383
25 Gosu Programs	385
The Structure of a Gosu Program	385
Metaline as First Line	385
Functions in a Gosu Program	386
Setting the Class Path to Call Other Gosu or Java Classes	386
26 Running Local Shell Commands	387
Running Command Line Tools from Gosu	387
27 Checksums	389
Overview of Checksums	389
Creating Fingerprints	390
How to Output Data Inside a Fingerprint	391
Extending Fingerprints	391
28 Properties Files	393
Reading Properties Files	393
29 Coding Style	395
General Coding Guidelines	395
Omit Semicolons	395
Type Declarations	395
The == and != Operator Recommendations and Warnings	395
Capitalization Conventions	397
Class Variable and Class Property Recommendations	397
Use 'typeis' Inference	397

About ClaimCenter Documentation

The following table lists the documents in ClaimCenter documentation.

Document	Purpose
<i>InsuranceSuite Guide</i>	If you are new to Guidewire InsuranceSuite applications, read the <i>InsuranceSuite Guide</i> for information on the architecture of Guidewire InsuranceSuite and application integrations. The intended readers are everyone who works with Guidewire applications.
<i>Application Guide</i>	If you are new to ClaimCenter or want to understand a feature, read the <i>Application Guide</i> . This guide describes features from a business perspective and provides links to other books as needed. The intended readers are everyone who works with ClaimCenter.
<i>Upgrade Guide</i>	Describes how to upgrade ClaimCenter from a previous major version. The intended readers are system administrators and implementation engineers who must merge base application changes into existing ClaimCenter application extensions and integrations.
<i>New and Changed Guide</i>	Describes new features and changes from prior ClaimCenter versions. Intended readers are business users and system administrators who want an overview of new features and changes to features. Consult the "Release Notes Archive" part of this document for changes in prior maintenance releases.
<i>Installation Guide</i>	Describes how to install ClaimCenter. The intended readers are everyone who installs the application for development or for production.
<i>System Administration Guide</i>	Describes how to manage a ClaimCenter system. The intended readers are system administrators responsible for managing security, backups, logging, importing user data, or application monitoring.
<i>Configuration Guide</i>	The primary reference for configuring initial implementation, data model extensions, and user interface (PCF) files. The intended readers are all IT staff and configuration engineers.
<i>Globalization Guide</i>	Describes how to configure ClaimCenter for a global environment. Covers globalization topics such as global locales, languages, date and number formats, names, currencies, addresses, and phone numbers. The intended readers are configuration engineers who work with locales and languages.
<i>Rules Guide</i>	Describes business rule methodology and the rule sets in ClaimCenter Studio. The intended readers are business analysts who define business processes, as well as programmers who write business rules in Gosu.
<i>Contact Management Guide</i>	Describes how to configure Guidewire InsuranceSuite applications to integrate with ContactManager and how to manage client and vendor contacts in a single system of record. The intended readers are ClaimCenter implementation engineers and ContactManager administrators.
<i>Best Practices Guide</i>	A reference of recommended design patterns for data model extensions, user interface, business rules, and Gosu programming. The intended readers are configuration engineers.
<i>Integration Guide</i>	Describes the integration architecture, concepts, and procedures for integrating ClaimCenter with external systems and extending application behavior with custom programming code. The intended readers are system architects and the integration programmers who write web services code or plugin code in Gosu or Java.
<i>Gosu Reference Guide</i>	Describes the Gosu programming language. The intended readers are anyone who uses the Gosu language, including for rules and PCF configuration.
<i>Glossary</i>	Defines industry terminology and technical terms in Guidewire documentation. The intended readers are everyone who works with Guidewire applications.

Conventions in This Document

Text style	Meaning	Examples
<i>italic</i>	Emphasis, special terminology, or a book title.	A <i>destination</i> 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 .
<code>monospaced</code>	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. In code blocks, bold formatting highlights relevant sections to notice or to configure.	Get the field from the <code>Address</code> object.
<code>monospaced italic</code>	Parameter names or other variable placeholder text within URLs or other code snippets.	Use <code>getName(first, last)</code> . <code>http://SERVERNAME/a.html</code> .

Support

For assistance with this software release, contact Guidewire Customer Support:

- At the Guidewire Resource Portal – <http://guidewire.custhelp.com>
- By email – support@guidewire.com
- By phone – +1-650-356-4955

Gosu Introduction

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

This topic includes:

- “Welcome to Gosu” on page 15
- “Running Gosu Programs and Calling Other Classes” on page 30
- “More About the Gosu Type System” on page 31
- “Gosu Case Sensitivity and Capitalization” on page 35
- “Gosu Statement Terminators” on page 35
- “Gosu Comments” on page 36
- “Gosu Reserved Words” on page 36
- “Gosu Generated Documentation (‘gosudoc’)” on page 37
- “Code Coverage Support” on page 37
- “Notable Differences Between Gosu and Java” on page 38
- “Get Ready for Gosu” on page 42

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*, which helps you find errors at compile time
- *imperative*
- *Java compatible*, which means you can use Java types, extend Java types, and implement Java interfaces
- *type inference*, which 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 34 for details.
- *XML/XSD support*
- *web service (SOAP) support*
- *an extensible type system*, which means that custom type loaders can dynamically inject types into the language. You can use these new types as native objects in Gosu. For example, custom type loaders dynamically add Gosu types for objects from XML schemas (XSDs) and from remote WS-I compliant web services (SOAP).
- *large companies around the world use Gosu every day in 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")
```

Gosu uses the Java type `java.util.String` as its native `String` type to manipulate texts. 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 28.

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 32.

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 31 and “Gosu Classes and Properties” on page 20.

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 35.

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 `NullPointerException` 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)  
}
```

Gosu has native support for *intervals*, which are sequences of values of the same type between a given pair of endpoint values. For instance, the set of integers beginning with 0 and ending with 10 is an integer interval. If it is a closed interval (contains the starting and ending values), it contains the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. The Gosu shorthand syntax for this is `0..10`. Intervals are particularly useful to write concise easy-to-understand `for` loops:

```
for( i in 1..10 ) {  
    print( i )  
}
```

You can optionally specify an open interval at one or both ends of the interval, meaning not to include the specified values. The Gosu syntax `1|..10` means an open interval on both sides, which means the values from 2 through 9.

Intervals do not need to represent numbers. Intervals can be a variety of types including numbers, dates, or other abstractions such as names. Gosu includes the built-in shorthand syntax (the two periods, shown earlier) for intervals of dates and common number types. You can also add custom interval types that support iterable comparable sequences. As long as your interval type implements the required interfaces, you can use your new intervals in `for` loop declarations:

```
for( i in new ColorInterval("red", "blue")) {  
    print( i )  
}
```

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

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

However, in practice the use of intervals makes most typical use of this pattern unnecessary, and you can use a Gosu `while` loop to duplicate this pattern.

To use intervals with `for` loops, they must be an iterative interval. You can choose to make custom non-iterative intervals if you want. They are mainly useful for math and theoretical work. For example, represent non-countable values like the infinite number of real numbers between two other real numbers.

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 231.

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 231. For more information about collections enhancement methods, many of which use blocks, see “Collections” on page 251.

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 Classes Implementing an Interface” on page 207.

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.

For more information, see “Enhancements” on page 227.

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 18).

```
// 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 19.

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 19.)
- You can also extend Java types and implement Java interfaces.

For more information, see “Calling Java from Gosu” on page 119.

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 24.

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  
}
```

Unlike Java, for special case usage you can optionally omit the type name entirely in a new expression if the type is known from its context. Do not omit the type name in typical code. However, for XML manipulation and dense hierarchical structure, it can make your Gosu code much easier to read and understand.

For more details, see “Creating and Instantiating Classes” on page 190 and “New Object Expressions” on page 73.

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 199.

Gosu also supports static methods, which means methods on the type rather than on object instances. See “Static Members” on page 23.

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 96.

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 23.

Define a property accessor function (a property `get`) 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/set/is Methods Convert to Gosu Properties” on page 121.

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.

There are additional null-safe operators. For example, specify default values with code like:

```
// Set display text to the String in the txt variable, or if it is null use "(empty)"  
var displayText = txt ?: "(empty)"
```

For more about Gosu null safety, see “Null Safety for Properties and Other Operators” on page 32.

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 198.

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 24.

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 34. 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 255.

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 (leftmost) 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 23.)

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, sometimes 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 `===(three equals signs)` operators:

Expression	Prints this Result	Description
<code>var x = 1 + 2 var s = x as String print(s == "3")</code>	<code>true</code>	These two variables reference the same value but different objects. If you use the double-equals operator, it returns <code>true</code> .
<code>var x = 1 + 2 var s = x as String print(s === "3")</code>	<code>false</code>	These two variables reference the same value but different objects. If you use the triple-equals operator, it returns <code>false</code> .

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 and Capitalization” on page 35.

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 89 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 89 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 215.

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 369

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.

Gosu supports named arguments syntax for annotations:

```
@MyAnnotation(a = "myname", b = true)
```

For more information, see “Annotations” on page 219.

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 347.

XML and XSD Support

Gosu provides 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>. For important information about using these APIs, see “Gosu and XML” on page 267.

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.

All the types from the XSD become native Gosu types, including element types and attributes. All these types appear naturally in the namespace defined by the part of the class hierarchy that you place the XSD. In other words, you put your XSDs side-by-side next to your Gosu classes and Gosu programs.

Suppose you put your XSD in the package directory for the package `mycompany.mypackage` and your XSD is called `mySchema.xsd`. Gosu lowercases the schema name because the naming convention for packages is lowercase. Gosu creates new types in the hierarchy:

```
mycompany.mypackage.myschema.*
```

For example, the following XSD file is called `driver.xsd`:

```
<xss:element name="DriverInfo">
  <xss:complexType>
    <xss:sequence>
      <xss:element ref="DriversLicense" minOccurs="0? maxOccurs="unbounded"/>
      <xss:element name="PurposeUse" type="String" minOccurs="0?/>
      <xss:element name="PermissionInd" type="String" minOccurs="0?/>
      <xss:element name="OperatorAtFaultInd" type="String" minOccurs="0?/>
    </xss:sequence>
    <xss:attribute name="id" type="xs:ID" use="optional"/>
  </xss:complexType>
</xss:element>
<xss:element name="DriversLicense">
  <xss:complexType>
    <xss:sequence>
      <xss:element name="DriversLicenseNumber" type="String"/>
      <xss:element name="StateProv" type="String" minOccurs="0?/>
      <xss:element name="CountryCd" type="String" minOccurs="0?/>
    </xss:sequence>
    <xss:attribute name="id" type="xs:ID" use="optional"/>
  </xss:complexType>
</xss:element>
```

The following Gosu code manipulates XML objects using XSD-based types:

```
uses xsd.driver.DriverInfo
uses xsd.driver.DriversLicense
uses java.util.ArrayList

function makeSampleDriver() : DriverInfo {
  var driver = new DriverInfo(){:PurposeUse = "Truck"}
  driver.DriversLicenses = new ArrayList<DriversLicense>()
  driver.DriversLicenses.add(new DriversLicense(){:CountryCd = "US", :StateProv = "AL"})
  return driver
}
```

For example, the following Gosu code uses an XSD called `demochildprops` to add two child elements and then print the results:

```
// create a new element, whose type is *automatically* in the namespace of the XSD
var e = new com.guidewire.pl.docexamples.gosu.xml.demochildprops.Element1()

// create a new CHILD element that is legal in the XSD, and add it as child
var c1 = new com.guidewire.pl.docexamples.gosu.xml.demochildprops.anonymous.elements.Element1_Child1()
e.addChild(c1)

// create a new CHILD element that is legal in the XSD (and which requires an int), and add it as child
var c2 = new com.guidewire.pl.docexamples.gosu.xml.demochildprops.anonymous.elements.Element1_Child2()
c2.$Value = 5 // this line automatically creates an XMLSimpleType -- but code is easy to read
e.addChild(c2)
```

Web Service Support (Consuming WSDL)

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.

For important information about using these APIs, see “Calling WS-I Web Services from Gosu” on page 71 in the *Integration Guide*.

The following example uses a hypothetical web service `SayHello`.

```
// -- get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.wsi.myservice.SayHello()

// -- set security options
service.Config.Http.Authentication.Basic.Username = "jms"
service.Config.Http.Authentication.Basic.Password = "b5"

// -- call a method on the service
var result = service.helloWorld()
```

Gosu Character Set

Because 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.

Running Gosu Programs and Calling Other Classes

To use Gosu, the initial file that you run **must** be a Gosu program. A Gosu program file has the `.gsp` file name extension. Gosu code in a program can call out to other Gosu classes and other types.

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 Gosu program (`.gsp` file) can call any necessary code, including Gosu or Java classes. If you want to mirror the Java style, your `.gsp` file can contain a single line that calls a `main` method on an important Gosu class or Java class.

To tell Gosu where to load additional classes, do either of the following:

- Use the `classpath` **argument** on the command line tool. See “Command Line Tool Options” on page 378.
- Add a `classpath` **statement** at the top of your Gosu program.

To use other Gosu classes, Java classes, or Java libraries:

1. Create a package-style hierarchy for your class files somewhere on your disk. For example, if the root of your files is `ClaimCenter/MyProject/`, put the class files for the Gosu class `com.example.MyClass` at the location `ClaimCenter/MyProject/com/example/MyClass.gs`.

2. In your Gosu program, tell Gosu where to find your other Gosu classes and Java classes by adding the classpath statement.

Typically you would place Java classes, Gosu classes, or libraries in 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 file for the class `mypackage.MyClass` to the location:

```
C:\gosu\myprograms\test1\src\mypackage\ MyClass.class
```

Copy your library files to locations such as:

```
C:\gosu\myprograms\test1\lib\mylibrary.jar
```

For this example, add two directories to the class path with the following Gosu statement:

```
classpath "src,lib"
```

For more information about Gosu programs, see “Gosu Programs” on page 385.

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 violations 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 and Other Gosu Editor Tools
- Type Usage Searching

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

- “Type System” on page 355
- “Basic Type Coercion” on page 356
- “Variable Type Declaration” on page 88.

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 34, or the full topic “Gosu Generics” on page 239. 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 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 and Other Gosu Editor Tools

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.

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 “Changing the Root Entity of a Rule” on page 28 in the *Rules Guide* for how to use Guidewire Studio find and replace functionality.

Null Safety for Properties and Other Operators

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.PropertyA.PropertyB.PropertyC
```

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`.

Gosu provides a variant of the period operator that is always explicitly null-safe for both property access and method access. The null-safe period operator has a question mark before it: `?.`

If the value on the left of the `?.` operator is `null`, the expression evaluates to `null`.

For example, the following expression evaluates left-to-right and contains three null-safe property operators:

```
obj?.PropertyA?.PropertyB?.PropertyC
```

Null Safe Method Calls

By default, 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:

```
house.myaction()
```

If `house` 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`.

In contrast, a *null-safe method call* does not throw an exception if the left side of the period character is `null`. Gosu just returns `null` from that expression. In contrast, using the `?.` operator calls the method with null safety:

```
house?.myaction()
```

If `house` is `null`, Gosu does not throw an exception. Gosu simply returns `null` from the expression.

Null-Safe Versions of Other Operators

Gosu provides other null-safe versions of other common operators:

- The null-safe default operator (`?:`). This operator lets you specify an alternate value if the value to the left of the operator is `null`. For example:

```
var displayName = Book.Title ?: "(Unknown Title)" // return "(Unknown Title)" if Book.Title is null
```
- The null-safe index operator (`?[]`). Use this operator with lists and arrays. It returns `null` if the list or array value is `null` at run time, rather than throwing an exception. For example:

```
var book = bookshelf?[bookNumber] // return null if bookshelf is null
```
- The null-safe math operators (`?+, ?-, ?*, ?/, and ?%`). For example:

```
var displayName = cost ?* 2 // multiply times 2, or return null if cost is null
```

See “Handling Null Values In Expressions” on page 83.

Design Code for Null Safety

Use null-safe operators where appropriate. They make code easy to read and easier to handle edge cases.

You can also design your code to take advantage of this special language feature. For example, expose data as *properties* in Gosu classes and interfaces rather than setter and getter methods.

See Also

- For more examples and discussion, see “Handling Null Values In Expressions” on page 83

IMPORTANT For more information about property accessor paths and designing your APIs around this feature, see “Handling Null Values In Expressions” on page 83.

Generics in Gosu

Generics are a way of abstracting behavior of a type to support working with multiple types of objects. Generics are particularly useful for implementing collections (lists, maps) in a type-safe way. At compile time, each use of the collection can specify the specific type of 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 specify a type, 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
```

This is called *parameterizing a generic type*. Read `ArrayList<Date>` in English as “an array list of date objects”.

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

The Gosu generics implementation is 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 239.

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 362 for details.

Gosu Case Sensitivity and Capitalization

Gosu is case-sensitive for most types. For example, if a type is declared as `MyClass`, you cannot type it as `myClass` or `myclass`.

All Guidewire entity types are case insensitive. However, it is best to write your code in a case-sensitive way.

There are standard conventions for how to capitalize different language elements. For example, local variable names have an initial lowercase letter. Type names, including class names have an initial uppercase letter. For more details, see “Capitalization Conventions” on page 397.

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 231). 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.PolicyNumber;cnum = Claim.ClaimNumber
```

Gosu Comments

Comment your Gosu code as you write 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 <i>commenting out</i> 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 “Generating Rule Debugging Information” on page 31 in the *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
- new
- as
- null
- break
- override
- case
- package
- catch
- private
- class
- property
- continue
- protected
- default
- public
- do
- readonly
- else
- request
- eval
- return

- except
- execution
- extends
- finally
- final
- find
- for
- foreach
- function
- get
- hide
- implements
- index
- interface
- internal
- native
- session
- set
- static
- super
- switch
- this
- try
- typeas
- typeis
- typeof
- unless
- uses
- var
- void
- while

Gosu Generated Documentation ('gosudoc')

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.mycompany.internal.*
```

Code Coverage Support

Code coverage tools analyze the degree of testing of programming code. For Gosu code in Studio, ClaimCenter supports code coverage tools that use Java class files as input to bytecode analysis. Gosu compiles to Java class files.

Direct your code coverage tools to the following directory:

`ClaimCenter/modules/configuration/out/classes`

This feature requires tools that use Java class files, not source code, as input.

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 25.	<code>a.B = c.D</code> B and D must exactly match the field declarations.	<code>a.B = c.D</code> Match the code capitalization to match the property declarations. Other capitalizations work, but are not recommended, such as: <code>a.b = c.d</code>	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)	<code>x = 1;</code>	<code>x = 1</code>	Optional
Print to console with the <code>print</code> function. For compatibility with Java code while porting to Gosu, you can optionally call the Java class <code>java.lang.System</code> .	<code>System.out.println("hello");</code>	<code>print("hello")</code> <code>uses java.lang.System System.out.println("hello world")</code>	Optional
For Boolean operators, optionally use more natural English versions. The symbolic versions from Java also work in Gosu.	<code>(a && b) c</code>	<code>(a and b) or c</code> <code>(a && b) c</code>	Optional
Null-safe property accessor paths (see "Property Accessor Paths are Null Safe" on page 23)	<code>// initialize to null ArrayList<Windows> x = null</code> <code>// check earlier parts of // path for null, to avoid // null pointer exception if(house != null and house.Walls != null) { x = house.Walls.Windows }</code>	<code>var x = house.Walls.Windows</code> <code>Required for cases in which you rely on throw- ing null pointer excep- tions</code>	Required for cases in which you rely on throw- ing null pointer excep- tions
Functions and Variables			
In function declarations:	<ul style="list-style-type: none">• use the keyword <code>function</code>• list the type after the variable, and delimited by a colon. This is true for both parameters and return types.	<code>public int addNumbers(int x, String y) { ... }</code>	<code>public function addNumbers(x : int, y : String) : int { ... }</code>
			Required

Difference	Java	Gosu	Required change?
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.	public String format(Object... args);	// function declaration public function format(args : Object[]) // method call using // initializer syntax 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.			
Gosu supports the unary operator assignment statements ++ and --. However: <ul style="list-style-type: none"> 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 89.	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			
For coercions, use the as keyword. Optionally, Gosu supports Java-style coercion syntax for compatibility.	int x = (int) 2.1 Optional, Gosu supports Java-style coercion syntax for compatibility.	// Gosu style var x = 2.1 as int // Java compatibility 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

Difference	Java	Gosu	Required change?
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 358.	Object x = "nice" Int s1 = 0 if(x instanceof String) { s1 = ((String) x).length }	var x : Object = "nice" var s1 = 0 if(x typeis String) { s1 = 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 <i>reified generics</i> . In contrast, Java removes this information (this is called <i>type erasure</i>).	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
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			

Difference	Java	Gosu	Required change?
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 <code>Iterable</code>). Optionally add “ <code>index indexVar</code> ” 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).	<pre>int[] numbers = {1,2,3}; for (int item : numbers) { // } }</pre>	<pre>var numbers : int[] = {1,2,3}; for (item in numbers) { // }</pre>	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.	<pre>for(int i=1; i<20; i++){ // }</pre>	<pre>for (item in 20) { // }</pre> <p>Using Gosu intervals:</p> <pre>for(i in 1..50) { print(i) }</pre> <p>verbose style:</p> <pre>var i = 0 while (i < 20) { // i++ }</pre>	Required
Gosu has native support for <i>intervals</i> , which are sequences of values of the same type between a given pair of endpoint values. For instance, the set of integers beginning with 0 and ending with 10 is the shorthand syntax <code>0..10</code> . Intervals are particularly useful to write concise easy-to-understand for loops.			
Gosu does not support the <code>for(initialize;compare;increment)</code> syntax in Java. However, you can duplicate it using a <code>while</code> statement (see example).			

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 227.	n/a	<pre>enhancement StrLenEnhancement : java.lang.String { public property get PrettyLength() : String { return "length : " + this.length() } }</pre>	Optional
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 231. Blocks can also be useful as a shortcut for implementing one-method interfaces (see “Blocks as Shortcuts for Anonymous Classes” on page 237).	n/a	<pre>\ x : Number -> x * x</pre>	Optional
Native XML support and XSD support. See “Gosu and XML” on page 267.	n/a	<pre>var e = schema.parse(xmlText)</pre>	Optional

Difference	Java	Gosu	Required change?
Native support for consuming web services with syntax similar to native method calls. See “Consuming WS-I Web Service Overview” on page 71 in the <i>Integration Guide</i> .	n/a	extAPI.myMethod(1, true, "c")	Optional
Native String templates and file-based templates with type-safe parameters. See “Gosu Templates” on page 347.	n/a	var s = "Total = \${ x }."	Optional
Gosu uses the Java-based collections APIs but improves upon them: <ul style="list-style-type: none"> 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 251.		<pre>// easy initialization var strs = {"a", "ab", "abc"}</pre> <pre>// array-like "set" and "get" strs[0] = "b" var val = strs[1]</pre> <pre>// new APIs on Java // collections types strList.each(\ str -> { print(str) })</pre>	Optional
List and array expansion operator. See “List and Array Expansion (*.*)” on page 255.	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.

Types

This topic describes common data types and how to use them in Gosu. For more information about manipulating types or examining type information at run time, see “Type System” on page 355.

This topic includes:

- “Access to Java Types” on page 43
- “Primitive Types” on page 44
- “Objects” on page 44
- “Boolean Values” on page 47
- “Sequences of Characters” on page 48
- “Array Types” on page 52
- “Numeric Literals” on page 55
- “Entity Types” on page 56
- “Typekeys and Typelists” on page 56
- “Compatibility with Earlier Gosu Releases” on page 58

Access to Java Types

Gosu is built on top of the Java language and runs within the Java Virtual Machine (JVM). Gosu automatically loads all Java types, so you have full direct access to Java types, such as classes, libraries, and primitive (non-object) types. Access to Java types includes:

- Instantiate Java classes with the `new` keyword
- Access to Java interfaces with the `new` keyword
- Call static methods on Java types
- Call object methods on instantiated objects
- Get properties from Java objects
- Support for Java primitive types

The most common Java object types maintain their fully-qualified name but are always in scope, so you do not need to fully qualify in typical code. For example, `Object`, `String`, and `Boolean`. In the case of ambiguity with similarly named types currently in scope, you must fully-qualify these types, such as `java.lang.String`. Other Java types are available but must be fully qualified, for example `java.util.ArrayList`.

Gosu includes transformations on Java types that make your Gosu code more easily understandable. For example turning getters and setters into Gosu properties. For details of Java-Gosu integration, see “Calling Java from Gosu” on page 119.

Primitive Types

Gosu supports the following Java primitive types, which are provided for compatibility with existing Java code. From Gosu you can access the Java object versions (non-primitives) of the Java primitive types. For example, `java.lang.Boolean` is an object type that wraps the behavior of the `boolean` primitive. Primitive types do not perform better in terms of performance or space compared to their object versions.

The following table compares primitive types and object types.

Type	Primitive types	Object Types
Extends from <code>Object</code> class		•
Can reference an object		•
A variable of this type could contain <code>null</code> at runtime		•
Exposes methods		•
Exposes properties		•
Cannot be a member of a collection	•	

The following table lists the Java primitives that you can access from Gosu. The table mentions IEEE 754, which is the Institute of Electrical and Electronics Engineers standard for Binary Floating-Point Arithmetic. For more information, refer to:

http://en.wikipedia.org/wiki/IEEE_754

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

For more information and related APIs, see “Working with Primitive Types” on page 363.

Objects

The root type for all object types in Gosu is the Java class `java.lang.Object`. An object encapsulates some data (variables and properties) and methods (functions that act on the object). Because it is always in scope, simply type `Object` unless a similarly-named type is in scope and requires disambiguation.

Do not create objects directly with the root object type `Object`. However, you can create classes that extend the root object type `Object`. If you do not declare a new class to extend a specific class, your new class extends the `Object` class.

In some cases you may need to declare a variable that use the type `Object` to support a variety of object subclasses. For example, you can define a collection to contain a heterogeneous mix of various object types, all of which extend the root object type `Object`.

Examples

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

To create or use objects, see “Object Instantiation” on page 45.

IMPORTANT To work with Guidewire entity objects, see “The ClaimCenter Data Model” on page 155 in the *Configuration Guide* and “Bundles and Database Transactions” on page 331.

For more information about the Gosu type system, see “Type System” on page 355.

Object Instantiation

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 object data and methods. To instantiate the class means to use the class definition to create an in-memory copy of the object with its own object data. Other code can get or set properties on the object. Other code can call methods on the object, which are functions that perform actions on that instance unique of the object.

In ClaimCenter, entity instances are special objects defined through the data model configuration files. ClaimCenter persists data in a database at special times in the object life cycle. For more information “Working with the Data Dictionary” on page 149 in the *Configuration Guide* and “Bundles and Database Transactions” on page 331.

Creating New Objects

You use the Gosu `new` operator to create an instance from a class definition or other type that can be instantiated.

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 73 for more details.

Object Property Assignment

Property assignment is similar to variable assignment.

Syntax

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

Some properties are write-protected. For example, the following Gosu code:

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

That code causes the following error:

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

Other properties are read-protected but can be written.

Example

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

Property Assignment Triggers Instantiation of Intermediate Objects

In general, if you try to assign a property on a variable, and the variable evaluates to `null` at runtime, Gosu throws a null pointer exception.

However, if you set a property using property path syntax containing at least two objects, in some cases Gosu can automatically instantiate an intermediate object in the path.

For example, suppose the following:

- You have Gosu classes called `AClass` and `BClass`.
- `AClass` has a property called `B` that contains a reference to an instance of class `BClass`.
- The `BClass` class has a property called `Name` that contains a `String` value.
- Some code has a variable called `a` that contains an instance of type `AClass`.

At run time, if `a.B` contains a non-null reference, it is easy to predict what the following code does:

```
a.B.Name = "John"
```

Gosu first evaluates `a.B`, and then on the result object sets the `Name` property to the value "John".

However, if the `AClass.B` property supports instantiation of intermediate objects, the same code works even if `a.B` is `null` at run time.

At run time, if Gosu detects that `a.B` is `null`:

1. Gosu instantiates a new instance of `BClass`
2. Gosu sets `a.B` to the new instance
3. Gosu sets `a.B.Name` property on it.

For all Guidewire entity types, properties that contain a foreign key reference support automatic instantiation of intermediate objects.

You can add instantiation of intermediate objects to any Gosu class property. On the line before the property declaration, add the line:

```
@Autocreate
```

Object Property Access

Gosu retrieves a property's value using the period operator. You can chain this expression with additional property accessors. For important details about how Gosu handles `null` values in the expression to the left of the period, see "Handling Null Values In Expressions" on page 83.

Syntax

```
object.PROPERTY_NAME
```

Examples

Expression	Result
<code>Claim.Contacts.Attorney.Name</code>	Some Name
<code>Claim.Addresses.State</code>	New Mexico

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. For more details about how Gosu handles null values in the expression to the left of the period, see “Handling Null Values In Expressions” on page 83.

Syntax

`object.METHOD_NAME()`

Example

Expression	Result
<code>claim.isClosed()</code>	Return a Boolean value indicating the status of Claim
<code>claim.resetFlags()</code>	Reset flags for this claim

See “Static Method Calls” on page 82 for more details. See also “Using Reflection” on page 360 for regarding using type information to determine methods of an object.

Boolean Values

From Gosu code, two types represent the values true and false:

- The Java primitive type `boolean`. Possible values are `true` and `false`.
- The Java `Boolean` object, which is an object wrapper around the primitive type. Possible values for variables declared to the `Boolean` data type are `true`, `false`, and `null`. The fully-qualified type name is `java.lang.Boolean`. Because it is always in scope, simply type `Boolean` unless a similarly-named type is in scope and requires disambiguation.

For both `boolean` and `Boolean`, some values can coerce to `true` or `false`. The following table describes coercion rules. The rightmost column indicates whether the coercion requires the `as` keyword, for example `1 as boolean`.

Value	Type of value	Coerces to	Explicit coercion is required	Comment
1	int	true	No	
0	int	false	No	
"true"	String	true	No	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".

				Remember that the <code>boolean</code> type is a primitive and can never contain <code>null</code> . However, the <code>Boolean</code> type is an object type, and variables with that type can contain <code>null</code> .
				Be careful to check for <code>null</code> values for variables declared as <code>String</code> to avoid ambiguity in how your code handles <code>null</code> values. A <code>null</code> value may indicate uninitialized data or other unexpected code paths.

Value	Type of value	Coerces to	Explicit coercion is required	Comment
"false" or any non-null String value other than "true"	String	false	No	See note for column value "true".
null (only possible for an object type)	see note	If coerced to the boolean type, the result depends on type of the declared variable. For some types including Object and number types such as Integer, null coerces to the value false. For other types, coercion is disallowed at compile time. If coerced to the Boolean type, value remains null.	No	See note for column value "true".

Notice some important differences between primitive and 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. Depending on the declared type of the variable, this coercion may be disallowed at compile time.

For important information about primitives and comparisons of boxed and unboxed types in Gosu, see “Working with Primitive Types” on page 363.

Example

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

Sequences of Characters

To represent a sequence of characters in Gosu, use the Java type `java.lang.String`. Because it is always in scope, simply type `String` unless a similarly-named type is in scope and requires disambiguation. Create a `String` object by enclosing a string of characters in beginning and ending double-quotation marks. Example values for the `String` data type are "hello", "123456", and "" (the empty string).

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. A variable declared to type `String` can hold the value `null`, the empty `String` (""), or a non-empty `String`.

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, `HasContent` returns `false` if either 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
```

Notice that whether the variable holds an empty `String` or `null`, the `HasContent` method returns `false`. This means that the `HasContent` method is more intuitive in typical cases where `null` represents absence of data.

String Properties in Entity Instances Have Special Setting Behavior

When you **set** a `String` data type property on an entity instance, there is special behavior that does not happen for `String` properties on classes. When you set a `String` value on a database-backed entity type property:

- Gosu removes spaces from the beginning and end of the `String` value, although this is configurable. See “Configuring Whitespace Removal for Entity Text Properties” on page 50.
- If the result is the empty `String` (“”), Gosu sets the value `null` instead of the empty `String`. This is not configurable

In both cases, this change happens immediately as you set the value. This change is not part of committing the data to the database. If you get the property value after setting it, the value is already updated.

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 48.

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 on a database-backed `String` property on an entity type. Set the `trimwhitespace` column parameter in the data model definition of the `String` column. For columns that you define as `type="varchar"`, Gosu trims 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`.

For both trimming and converting empty `String` values to `null`, the change happens immediately as you set the value.

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
```

Type `"new String()` into the Gosu Scratchpad and then press period to see the full list of methods.

String Utilities

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

For example, instead of the Java native method `replace` on `java.lang.String`, to perform search and replace you can use the `StringUtil` method `substituteGlobalCaseInsensitive`.

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 347.

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 347.
\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 347.

Keys of Guidewire Entity Instances

`Key` is a Guidewire data model entity base type that uniquely identifies an entity instance. The internal ID of a `Claim` entity instance (`Claim.Id`) has type `Key`. Sometimes functions or internal domain methods require `Key` objects as parameters, so it is often useful to cast an entity instance to a key.

Example

```
var id = myEntityInstance as Key
```

Do not confuse a key (the type called `Key`) and a typekey (the type called `Typekey`). See “Typekeys and Type-lists” on page 56.

Array Types

An *array* is a collection of data values, with each element of the array associated with a number or *index*. In typical Gosu code, simply use angle brackets after the type name, such as `String[]` to represent an array of `String` objects. Use a zero-based index number to access an array member.

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, all examples contain exactly four members
var s1 = new String[4]
var s2 = new String[] {"This", "is", "a", "test"}

// arrays of integers, all examples contain exactly three members
var int1 = new int[3]
var int2 = new int[] {1,2,3}
var int3 : int[] = {1,2,3}
```

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

Syntax

`EXPRESSION[INDEX_VALUE]`

Examples

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

You can iterate through the members of an array using a `for` loop. See “Iteration in For Statements” on page 93 for details.

You can create a new array with a default value for each array member using an included Gosu enhancement on the `Arrays` object. Call the `makeArray` method and pass 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 following code:

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

List Access Using Array Index Notation

In Gosu, you can Java language list members using standard array index notation. For much more information about Java lists and other collections in Gosu, see “Collections” on page 251. Also see “For Statements” on page 93 for examples of lists with array-style access syntax.

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 many situations, it is best to use collections such as `java.util.List` or `java.util.Map` rather than to use arrays. The `List` and `Map` classes inherit from the Java language although Gosu adds additional enhancement methods. For Gosu APIs related to using lists, maps, and other collections, see “Collections” on page 251.

Array Expansion

Gosu supports an operator that expands arrays and lists: the `*.` operator. For more information, see “List and Array Expansion (`*.`)” on page 255.

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.

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 retireable, 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.

Associative Array Syntax for Property Access

For most types, you can use *associative array syntax* to access properties of the object. Instead of using a numeric index surrounded by brackets, use a `String` value containing the property name. This syntax is useful if the property name is unknown at compile time, but can be determined at run time with user-entered data.

WARNING Using associative array syntax, Gosu cannot check at compile time that you correctly typed the property name for that type. Be careful with this feature to prevent unexpected run time errors. Use typesafe property access instead where possible.

The following simple example shows associative array syntax:

```
var str = "PostalCode"
var pc = myContact[str]
```

Associative arrays are similar to using the Java map class `java.util.Map`, which Gosu supports, in that you can use a non-numeric key to a value. For more information about maps and other collections, see “Collections” on page 251.

The associative array syntax for property access works with most classes, including the `Map` class and any object type that does not naturally take a numeric array-style index. The `java.lang.String` class does not work with the associative array syntax. A `String` object behaves as an ordered list of characters in Gosu, so it requires a numeric array-style index.

Accessing property information at run time is part of a programming feature called *reflection*. There are other APIs for reflection. For more information, see “Using Reflection” on page 360.

If the property name does not exist at run time, Gosu throws an exception. 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
    }

```

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
<code>person["StreetAddress"]</code>	"123 Main Street"	example of a single associative array access
<code>person["Address"]["City"]</code>	"Birmingham"	example of a double associative array access. This is equivalent to the code <code>person.Address.City</code> .

You can use associative array syntax to set properties and values as well as get them. For example:

```

var address : Address
var city = address["City"]      // get a property
address["City"] = "San Mateo"   // set a property

```

Legacy Array Type

For compatibility with prior Gosu versions, there is type called `Array`. Do not use that for new code. Instead use the `TYPE_NAME[]` syntax with a specific type name.

Numeric Literals

Gosu natively supports numeric literals of the most common numeric types, as well as a binary and hex syntax. Gosu uses the syntax to infer the type of the object.

For example:

- Gosu infers that the following variable has type `BigInteger` because the right side of the assignment uses a numeric literal `1bi`. That literal means “1, as a big integer”
`var aBigInt = 1bi`
- Gosu infers that the following variables have type `Float` because the right side of the assignment uses a numeric literal with an `f` after the number.
`var aFloat = 1f`
`var anotherFloat = 1.0f`
- For numeric literals that have a suffix, you can omit the suffix if the type is declared explicitly such that no type inference is necessary.

The following table lists the suffix or prefix for different numeric, binary, and hexadecimal literals.

Type	Syntax	
byte	suffix: b or B	<code>var aByte = 1b</code>
short	suffix: s or S	<code>var aShort = 1s</code>
int	none	<code>var anInt = 1</code>
long	suffix: l (lowercase L) or L	<code>var aLong = 1L</code>
float	suffix: f or F	<code>var f1 = 1f</code> <code>var f2 = 1.0f</code>
double	suffix: d or D	<code>var aDouble = 1d</code>
<code>BigInteger</code>	suffix: bi or BI	<code>var aBigInt = 1bi</code>
<code>BigDecimal</code>	suffix: bd or BD	<code>var aBigD = 1bd</code> <code>var anotherBigD = 1.0bd</code>

Type	Syntax	
int	prefix: 0b or 0B	<code>var maskVal1 = 0b0001 // 0 and 1 only</code> <code>var maskVal2 = 0b0010</code> <code>var maskVal3 = 0b0100</code>
int as hexadecimal	prefix: 0x or 0X	<code>var aColor = 0xFFFF // 0 through F only</code>

Entity Types

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 “Working with the Data Dictionary” on page 149 in the *Configuration Guide*.

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

For example, to pass an `Address` to an API that takes an `IEntityType` as the method argument, 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 and Typelists

Many Guidewire entity data model entities contain properties that contain typekeys. A typekey is similar to an enumeration. A typelist encapsulates a list of typekeys. Each typekey has a unique immutable code as a `String`, in addition to a user-readable (and localized) name. The code for a typekey is sometimes referred to as a typecode.

For example, the typelist called `AddressType` contains three typekey values. The typekeys have the codes `BUSINESS`, `HOME`, and `OTHER`.

For information about the ClaimCenter data model, see “The ClaimCenter Data Model” on page 155 in the *Configuration Guide*.

Typelist Literals

In most cases, to get a literal for a typelist, simply type the typelist 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, suppose you want to select from the `AddressType` typelist a reference to the typekey 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: `TYPENAME.TC_TYPECODENAME`. For example, `State.TC_CA`. That is the preferred approach.

If the programming context accepts a specific typekey type, as an alternate syntax you can use implicit coercion from the typekey's code as a `String` to the typekey instance itself. For example:

```
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 `TYPENAME.TC_TYPECODENAME` syntax to avoid confusing unexpected 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 not typesafe if the programming context accepts 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` containing 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 typecode against. In such cases, instead use the verbose syntax `TYPENAME.TC_TYPECODENAME`.

The 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 unknown at compile time, 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 a run-time 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 typekeys that are marked as *retired* (obsolete). 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 <code>String</code> 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

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
```

Compatibility with Earlier Gosu Releases

The following subtopics describe supported types that exist mainly for compatibility with earlier Gosu releases.

DateTime

The `DateTime` type encapsulates calendar dates or time (clock) values, or both.

IMPORTANT The `DateTime` class exists for compatibility with earlier Gosu releases. For new code, instead use the Java class `java.util.Date`.

The following table lists the supported formats for date and time definitions.

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

Individual characters in the previous table have the following meaning:

Character	Meaning
<code>a</code>	AM or PM (determined from 24-hour clock)
<code>d</code>	day
<code>E</code>	Day in week (abbreviation)
<code>h</code>	hour (24 hour clock)
<code>m</code>	minute
<code>M</code>	month
<code>s</code>	second
<code>S</code>	fraction of a second
<code>T</code>	parse as time (ISO8601)
<code>y</code>	year
<code>z</code>	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)

Initializing DateTime Values

If you declare and initialize a `DateTime` object, Gosu sets the value to the date and time at which it was allocated, measured to the nearest millisecond.

```
var rightNow = new DateTime() // Get a timestamp for the current date and time.
```

Initialize a DateTime Object to a Specific Date

You can declare a `DateTime` variable and initialize it to a specific date.

There is a constructor to initialize a `DateTime` value using a `String` value with the supported formats described earlier in this topic:

```
var d = new DateTime("Jun 3, 2005")
```

There is another constructor takes three arguments:

- **Year** – The year minus 1900
- **Month** – The month between 0-11
- **Day** – The day of the month between 1-31

For example, the following Gosu code declares and initializes a `DateTime` variable to January 1, 2012, with the time portion of the value set to 12:00 a.m. midnight (00:00:00.000).

```
var aDate = new DateTime(112, 0, 1) // Initialize a DateTime to January 1, 2012, 12:00 a.m. midnight.
```

Initialize a `DateTime` Object to a Special Date

You can declare a `DateTime` variable and initialize it to a well-known date: today, tomorrow, or yesterday. Also, you can use these well-known dates in expressions without declaring them as variables. The time component of these well-known dates is 12:00 a.m. midnight (00:00:00.000). The `DateTime.CurrentDate` property provides a current timestamp. It is equivalent to initializing a `DateTime` variable with the new `DateTime()` constructor.

For example, the following Gosu code declares and initializes a `DateTime` variable to the well-known date `Today`.

```
var dateToday = DateTime.Today
```

The following Gosu code uses the date `Yesterday` in a Boolean expression.

```
if (gw.api.util.DateUtil.compareIgnoreTime(aTask.CreateTime, DateTime.Yesterday == 0) {
    print ("You were assignt the task " + aTask.DisplayName + "yesterday.")
}
```

Working with `DateTime` Values

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 Scratchpad, and then press period to see the full list of methods. For example

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

Gosu `DateTime` and Java

The `DateTime` class exists for compatibility with earlier Gosu releases. For new code, instead use the Java class `java.util.Date`. Gosu represents `DateTime` objects internally using `java.util.Date`. However, this is internal only and thus it is not possible to access the `java.util.Date` type directly from a `DateTime` object.

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

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

Gosu's `DateTime` type exposes some functionality of the `Date` class using Gosu operators:

- Gosu supports the `Date` methods `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 `as` keyword, which is 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
```

Number

The `Number` data type represents all numbers, including integers and floating-point values.

IMPORTANT The `Number` class exists for compatibility with earlier Gosu releases. For new code, instead use the Java class `java.lang.Double`.

Possible values for the `Number` data type include:

- 1
- 246
- 3.14159
- NaN (represents not a number)
- Infinity (represents infinity)
- null

If you cast a `String` value to `Number`, if the `String` contains only number characters, the result is a number.

Array

The `Array` type exists for compatibility with earlier Gosu releases. For new code, instead use standard array syntax with bracket notation with a specific type, such as `Integer[]`. For more information, see “[Array Types](#)” on page 52.

Gosu Operators and Expressions

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

This topic includes:

- “Gosu Operators” on page 63
- “Standard Gosu Expressions” on page 65
- “Arithmetic Expressions” on page 65
- “Equality Expressions” on page 68
- “Evaluation Expressions” on page 70
- “Existence Testing Expressions” on page 70
- “Logical Expressions” on page 71
- “New Object Expressions” on page 73
- “Relational Expressions” on page 76
- “Unary Expressions” on page 78
- “Importing Types and Package Namespaces” on page 79
- “Conditional Ternary Expressions” on page 80
- “Special Gosu Expressions” on page 82
- “Handling Null Values In Expressions” on page 83

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.

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	<ul style="list-style-type: none"> • -3 • typeof "Test" • new Array[3]
binary	2	<ul style="list-style-type: none"> • 5 - 3 • a and b • 2 * 6
ternary	3	<ul style="list-style-type: none"> • 3*3 == 9 ? true : false

See also

“Operator Precedence” on page 64

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.

Operator	Description
.	Property access, array indexing, function calls and expression grouping. The operators with the question marks are the null-safe operators. See “Handling Null Values In Expressions” on page 83.
[]	
()	
?.	
?[]	
?:	
new	Object creation, object reflection
,	Array value list, as in {value1, value2, value3} Argument list, as in (parameter1, parameter2, parameter3)
as	
typeas	As, typeas
+	Unary operands (positive, negative values)
-	
~	Bit-wise OR, logical NOT, typeof, eval(expression)
!	
not	
typeof	
eval	
typeis	Typeis
*	Multiplication, division, modulo division
/	
%	
<<	Bitwise shifting
>>	
>>>	
+	Addition, subtraction, string concatenation. The versions with the question marks are the null-safe versions. See “Null-safe Math Operators” on page 85.
-	
?+	
?-	
<	Less than, less than or equal, greater than, greater than or equal
<=	
>	
>=	

Operator	Description
<code>==</code>	Equality, inequality. For general discussion and also comparison of <code>==</code> and <code>==></code> , see “Equality Expressions” on page 68.
<code>!=</code>	
<code>&</code>	bitwise AND
<code>^</code>	bitwise exclusive OR
<code> </code>	bitwise inclusive OR
<code>&&</code>	Logical AND, the two variants are equivalent
<code>and</code>	
<code> </code>	Logical OR, the two variants are equivalent
<code>or</code>	
<code>? :</code>	Conditional (ternary, for example, <code>3*3 == 9 ? true : false</code>)
<code>= += -= *= /= %= &= ^= = <<= >>=</code>	Assignment operator statements. These are technically Gosu statements, not expressions. For more information, see “Gosu Variables” on page 88.
<code>>>>=</code>	

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 357)
- Type Checking Expressions (see “Basic Type Checking” on page 357)

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
<code>3 + 5</code>	8
<code>8 + 7.583</code>	15.583
<code>"Auto" + "Policy"</code>	<code>"AutoPolicy"</code>
<code>10 + "5"</code>	<code>"105"</code>
<code>"Number " + 1</code>	<code>"Number 1"</code>

For the null-safe version of this operator, see “Null-safe Math Operators” on page 85.

Subtraction Operator (-)

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

Expression	Result
<code>9 - 2</code>	7
<code>8 - 3.359</code>	4.641
<code>"9" - 3</code>	6

For the null-safe version of this operator, see “Null-safe Math Operators” on page 85.

Multiplication Operator (*)

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

Expression	Result
<code>2 * 6</code>	12
<code>12 * 3.26</code>	39.12
<code>"9" * "3"</code>	27

For the null-safe version of this operator, see “Null-safe Math Operators” on page 85.

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
<code>10 / 2</code>	5
<code>5 / "2"</code>	2.5
<code>5 / "test"</code>	<code>Nan</code>
<code>1 / 0</code>	<code>Infinity</code>

Expression	Result
0 / 0	NaN
0/1	0

For the null-safe version of this operator, see “Null-safe Math Operators” on page 85.

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

For the null-safe version of this operator, see “Null-safe Math Operators” on page 85.

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. 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. 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
- Inequality 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.

Syntax

`a == b`

Examples

Expression	Result
<code>7 == 7</code>	<code>true</code>
<code>"3" == 3</code>	<code>true</code>
<code>3 == 5</code>	<code>false</code>

== 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, sometimes 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
<code>print("3" == "3")</code>	<code>true</code>	The two <code>String</code> objects contain the same value.
<code>print("3" == "4")</code>	<code>false</code>	The two <code>String</code> objects contain different values.
<code>print("3" === "4")</code>	<code>false</code>	Gosu represents the two <code>String</code> literals as separate objects in memory (as well as separate values).
<code>var x = 1 + 2</code> <code>var s = x as String</code> <code>print(s == "3")</code>	<code>true</code>	These two variables reference the same value but different objects. If you use the double-equals operator, it returns <code>true</code> .
<code>var x = 1 + 2</code> <code>var s = x as String</code> <code>print(s === "3")</code>	<code>false</code>	These two variables reference the same value but different objects. If you use the triple-equals operator, it returns <code>false</code> .
<code>print("3" === "3")</code>	<code>true</code>	This example is harder to understand. By just looking at the code, it seems like these two <code>String</code> objects would be different objects. However, in this case, the Gosu compiler detects they are the same <code>String</code> at compile time. Gosu optimizes the code for both usages of a <code>String</code> literal to point to the same object in memory for both usages of the "3".

Inequality Operator (!=)

The `!=` operator tests for relational inequality. The operands can be of any compatible types. The result is always `Boolean`.

Syntax

`a != b`

Examples

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

See also

For another use of the `!=` operator, see the examples in “Logical NOT” on page 72.

Evaluation Expressions

The `eval()` expression evaluates Gosu source at run time, 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
<code>eval("2 + 2")</code>	4
<code>eval(3 > 4 ? true : false)</code>	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.

There are two basic ways to use an `exists` expression:

- **Exists expressions in general use** – Gosu iterates across an array or a list but does not generate nor execute a SQL query. Consider exists expressions in general purpose Gosu code 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.
- **Exists expressions in legacy find expressions.**

IMPORTANT Find queries are a legacy style of querying a Guidewire database. For new code, use the query builder APIs. For more information, see “Query Builder APIs” on page 125.

Gosu optimizes the database query of a find expression 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 179 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
<code>exists (var e in Claim.Exposures where e == null)</code>	true or false
<code>exists(var group in Claim.Exposures.AssignedGroup where group.GroupType == "autofasttrack")</code>	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.

See also

For important differences between the types `Boolean` and `boolean`, as well as differences in coercion rules, see “Boolean Values” on page 47.

Supported Logical Operators

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 determines 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 determines 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
<code>(4 > 3) and (3 > 2)</code>	<code>(true/true) = true</code>
<code>(4 > 3) && (2 > 3)</code>	<code>(true/false) = false</code>

Expression	Result
(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 use (or 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** – Use the following approach for 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 357.

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.

Although it is often used as an expression, it can also optionally be a statement if the return value is unneeded. See “New Is Optionally a Statement” on page 88.

Syntax For Typical Cases

```

new javaType (argument_list)           // The optional argument list contains constructor arguments.  

new gosuType (argument_list)          // The optional argument list contains constructor arguments.  

new arrayType [size]  

new arrayType [] {array_Value_List}   // This syntax allows declaring a list of initial array members

```

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, or bundle. Passing no arguments is the best approach, generally. 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 Database Transactions” on page 331.

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 <code>new</code> 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 Database Transactions” on page 331.

Examples

Expression	Result
<code>new java.util.HashMap(8)</code>	Creates an instance of the <code>HashMap</code> Java class.
<code>new String[12]</code>	Creates a <code>String</code> array with 12 members with no initial values.
<code>new String[] {"a", "b", "c"}</code>	Creates a <code>String</code> array with three members, initialized to "a", "b", and "c".
<code>new Note()</code>	Constructs a new <code>Note</code> 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.
<code>new Note(myClaim)</code>	Constructs a new <code>Note</code> entity given a <code>Claim</code> as the bundle to use. If changes to the <code>Claim</code> submit to the database, the new note adds at the same time.
<code>new Note(myClaim.bundle)</code>	Constructs a new <code>Note</code> entity given a <code>Claim</code> as the bundle to use. If changes to the <code>Claim</code> submit to the database, the new note adds at the same time.

Optionally Omit Type Name with the new Keyword When Type is Determined From Context

If the type of the object is determined from the programming context, you can omit the type name entirely in the object creation expression with the `new` keyword.

WARNING Omitting the type name with the `new` keyword is strongly discouraged in typical code. Omit the type name only for XML manipulation and dense hierarchical structures with long type names. Types imported from XSDs sometimes have complex and hard-to-read type names. The following examples are simple only to make it easier to understand, not to promote usage in simple cases.

For example, first declare a variable with an explicit type. Next, assign that variable a new object of that type in a simple assignment statement that omits the type name:

```
// Declare a variable explicitly with a type.
var s : String

// Create a new empty string.
s = new()
```

You can also omit the type name if the context is a method argument type:

```
class SimpleObj {
}

class Test {
    function doAction (arg1 : SimpleObj) {
    }
}

var t = new Test()

// The type of the argument in the doAction method is predetermined,
// therefore you can omit the type name if you create a new instance as a method argument.
t.doAction(new())
```

The following example uses both local variables and class variables:

```
class Person {
    private var _name : String as Name
    private var _age : int as Age
}

class Tutoring {
    private var _teacher : Person as Teacher
    private var _student : Person as Student
}

// Declare a variable as a specific type to omit the type name in the "new" expression
// during assignment to that variable.
var p : Person
var t : Tutoring
p = new() // type name omitted
t = new() // type name omitted
```

```
// If a class var or other data property has a declared type, optionally omit the type name.
t.Teacher = new()
t.Student = new()
```

Object Initializer Syntax

Object initializers let you set properties on newly created objects immediately after new expressions. Use object initializers for compact and clear object declarations. They are especially useful if combined with data structure syntax and nested objects.

Simple Object Initializers

A simple object initializer looks like the following:

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

Object initializers comprise one or more property initializer expressions, separated by commas, and enclosed by curly braces. A property initializer expression is the following in order: a colon (:), a property name, an equals symbol (=), and a value (any expression that results in a value).

```
:propertyName = value
```

For example, suppose you have the following code, which sets properties on a new file container by using assignment statements that follow the new statement:

```
var myFileContainer = new my.company.FileContainer() // Create a new file container.

myFileContainer.DestFile = jarFile // Set the properties on the new file container.
myFileContainer.BaseDir = dir
myFileContainer.Update = true
myFileContainer.WhenManifestOnly = ScriptEnvironment.WHEN_EMPTY_SKIP
```

The following sample code is functionally equivalent to the preceding code but uses an object initializer to make the assignments within the bounds of in the new statement:

```
var myFileContainer = new my.company.FileContainer() { // Create a new file container,
    :DestFile = jarFile, :BaseDir = dir, :Update = true, // and set its properties at creation time.
    :WhenManifestOnly = ScriptEnvironment.WHEN_EMPTY_SKIP
}
```

Nested Object Initializers

You can use object initializers to create and initialize properties on nested objects within new statements. If an object property in a property initializer expression is itself an object, the value you assign can be a new object with its own object initializer.

For example, suppose you have the following code:

```
var testSet = new TestSet() // Create a test set.
testSet.name = "Root" // Set the name of the test set.

var test = new Test() // Create and initialize a test.
test.name = "test1"

testSet.tests.add(test) // Add the new test to the array of the test set.
testSet.tests.add(new Test()) // Create another test and add it to the array.
testSet.tests.get(1).final = true // Set the final property on the second test in the array.
testSet.tests.get(1).type = new TestType() // Create a test type and assign the type property
// on the second test in the array to it.

var testStyle = new TestStyle() // Create and initialize a test style.
testStyle.color = Red; // Gosu infers which enum class is appropriate!
testStyle.number = 5

testSet.style = testStyle // Set the style property of the test set.

return testSet.toXML() // Convert the test set to XML.
```

You can rewrite the preceding code by using nested new object expressions with their own object initializers to reflect visually the nested object structure that the code creates.

```
var testSet = new TestSet() {
    :name = "Root", // Create a test set.
    :tests = { // Set the name of the test set.
```

```

new Test() { :name = "test1"},           // Create and initialize a test and add it to the array.
new Test() {                           // Create another test and add it to the array
    :final = true,                   // Set the final property to true on the second test.
    :type = new TestType()          // Create a test type and set the type property
},                                     // on the second test in the array to it.
:style =                                // Create and initialize a test style and set the
new TestStyle() {                         // style of the test set to it.
    :color = Red,                      // Color of the test
    :number = 5                        // Number of the test
}
}

```

Nested object initializers are especially useful when constructing in-memory XML data structures.

Special Syntax for Initializing Lists, Collections, and Maps

There are specialized initializer syntax and rules for creating new lists, collections, and maps. For more information, “Basic Lists” on page 251 and “Basic Hash Maps” on page 253.

Referencing Existing Guidewire Entities

If you want to reference an already-existing Guidewire business data entity instance, do not use the `new` keyword. Instead, do a query for the entity instances that you care about. For more information, see “Query Builder APIs” on page 125 for details.

There is an alternate form to look up an entity instance, although generally not 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 relational 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

Expression	Result
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 run time, 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 the type to which an expression evaluates. 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 357.

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 (`!`), 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 (`~`) 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()
```

Note: 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 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 following packages:

- `entity.*`
- `pcf.*`
- `pcftest.*`
- `perm.*`
- `productmodel.*`
- `snapshot.*`
- `soap.*` – only if SOAP type loader is available
- `typekey.*`
- `xsd.*` – only if XML/XSD type loader is available

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

A conditional ternary expression uses the `Boolean` value of one expression to decide which of two other expressions to evaluate. A question mark (?) separates the conditional expression from the alternatives, and a colon (:) separates the alternatives from each other. In other programming languages, ternary expressions are sometimes known as using the *conditional operator* or *ternary operator*, one which has three operands instead of two.

Syntax

```
conditionalExpression ? trueExpression : falseExpression
```

The second and third operands that follow the question mark (?) must be of compatible types.

At run time, Gosu evaluates the first operand, the conditional expression. If the conditional expression evaluates to `true`, Gosu evaluates the second operand, the expression that follows the question mark. It ignores the third operand, the expression that follows the colon. Conversely, if the conditional expression evaluates to `false`, Gosu ignores the second operand, the true expression, and evaluates the third operand, the false expression.

For example, consider the following ternary expression:

```
myNumberVar > 10 ? print("Bigger than 10") : print("10 or less")
```

At run time, if the value of `myNumberVar` is greater than 10, Gosu prints “Bigger than 10”. Conversely, if the value of `myNumberVar` is 10 or less, Gosu prints “10 or less”.

Examples

Ternary expression	Evaluation result
<code>3 > 4 ? true : false</code>	<code>false</code>
<code>3*3 == 9 ? true : false</code>	<code>true</code>

Ternary Expression Types at Run Time and Compile Time

At run time, the type of a ternary expression is the type of the true expression or the false expression, depending on how the conditional expression evaluates. Often with ternary expressions, the true and false expressions evaluate to the same type, but not always.

For example, consider the following ternary expression.

```
var ternaryResult = aContact.Status == "new" ? "hello" : false
```

In the example, the true expression is of type `String` and the false expression is of type `Boolean`. If at run time the contact is `new`, `ternaryResult` is of type `String`, and its value is `hello`. Conversely, if the contact is not `new`, `ternaryResult` is of type `Boolean`, and its value is `false`. Although the true expression and the false expression are of different types, their types are compatible because `String` and `Boolean` descend from `Object`.

At compile time, if the true and false expressions are of different types, Gosu reconciles the type of the ternary expression to the type of their nearest common ancestor. Gosu requires that the types of the true and false expres-

sions in a ternary expression be compatible so Gosu can reconcile their types. If they have no common ancestor, the type at compile time of the ternary expression is `Object`.

For example, reconsider the earlier example.

```
var ternaryResult = aContact.Status == "new" ? "hello" : false
```

At compile time, Gosu sets the type of `ternaryResult` to `Object`. Because Gosu implicitly declares its type as `Object`, the `ternaryResult` variable can hold instances of type `String` and of type `Boolean`. The following example makes the type as set by the compiler explicit.

```
// The type of a ternary expression is the common ancestor type of its true and false expressions.  
var ternaryResult : Object = aContact.Status == "new" ? "hello" : false
```

At run time, the type evaluation of the ternary expression and the `ternaryResult` variable varies depending on the current state of the system. Several different type checking keywords produce varying results. For example, if the contact in the following example is new, `ternaryResult` is of type `String`, although its static type remains `Object`.

```
var ternaryResult = aContact.Status == "new" ? "hello" : false // Contact is new.  
  
print(ternaryResult typeis Object)  
print(ternaryResult typeis String)  
print(ternaryResult typeis Boolean)  
print(typeof ternaryResult)  
print(staticTypeOf ternaryResult)
```

The preceding example produces the following output.

```
true  
true  
false  
String  
Object
```

Conversely, if the contact in the following example is not new, `ternaryResult` is of type `Boolean`, although its static type remains `Object`.

```
var ternaryResult = aContact.Status == "new" ? "hello" : false // Contact is not new.  
  
print(ternaryResult typeis Object)  
print(ternaryResult typeis String)  
print(ternaryResult typeis Boolean)  
print(typeof ternaryResult)  
print(staticTypeOf ternaryResult)
```

The preceding example produces the following output.

```
true  
false  
true  
Boolean  
Object
```

Primitive Type Coercion and Ternary Expressions

If the true or false expression in a ternary expression is of a primitive type, such as `int` or `boolean`, Gosu first coerces the primitive type to its boxed version. Then, Gosu searches the type hierarchy for a common ancestor type. For example, Gosu coerces the primitive type `boolean` to its boxed type `Boolean`.

Recursive Use of Ternary Expressions

Gosu supports recursive use of ternary expressions. The second and third operands, the true and false expressions, can themselves be ternary expressions. The ternary operator is syntactically right-associative.

For example, the recursive ternary expression `a ? b : c ? d : e ? f : g` evaluates with the explicit order of precedence `a ? b : (c ? d : (e ? f : g))`. At run time, the ternary expression reduces itself to one of the expressions `b`, `d`, `f`, or `g`.

See also

- For details of Boolean logic evaluation, see “Logical Expressions” on page 71.
- For more about type coercion and type checking, see “Type System” on page 355.

- For more about primitive and boxed types, see “Working with Primitive Types” on page 363.

Special Gosu Expressions

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

- Static Method Calls
- Function Calls
- Static Property Paths
- Handling Null Values In Expressions

Function Calls

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

Syntax

functionName(argumentList)

Examples

Expression	Result
<code>now()</code>	Current Date
<code>concat("limited-", "coverage")</code>	"limited-coverage"

Static Method Calls

Gosu uses the following syntax to call a static method on a type.

Syntax

typeExpression.staticMethodName(argumentList)

Examples

Expression	Result
<code>Person.isAssignableFrom(type)</code>	true/false
<code>java.lang.System.currentTimeMillis()</code>	Current time
<code>java.util.Calendar.getInstance()</code>	Java Calendar

For more information about static methods and the `static` operator, see “Modifiers” on page 198

Static Property Paths

Gosu uses the dot-separated path rooted at a Type expression to retrieve the value of a static property.

Syntax

TypeExpression.StaticProperty

Examples

Expression	Result
<code>Claim.TypeInfo</code>	<code>ClaimTypeInfo</code>

Expression	Result
LossCause.TC_HAIL	"hail" typekey
java.util.Calendar.FRIDAY	Friday value

For more information about static properties in classes, see “Modifiers” on page 198.

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.]typeName  
[typekey.]typeName
```

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

Handling Null Values In Expressions

Null-safe Property Access

A property path expression in Gosu is a series of property accesses in series, for example `x.P1.P2.P3`. There are two different operators you can use in Gosu to get property values:

- The standard period operator “.”, which can access properties or invoke methods. The standard period operator is only null-safe for properties, and not for method invocations.
- The null-safe period operator “?.”, which can access properties or invoke methods in a null-safe way. For properties, this is the same as the standard period operator. For methods you must this operator if you want a null-safe method invocation.

How the Standard Period Operator Handles Null

The standard “.” operator has a special behavior that some languages do not have. If any object property in a property path expression evaluates to `null`, in Gosu 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. This feature is called *null-safe shortcircuiting* for a property path expression.

For example, suppose that you have an expression similar to the following:

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

Remember that if any element in the path evaluates to `null`, the entire expression evaluates to `null`. If `claim` is `null`, the result is `null`. Also, if `claim.AssignedGroup` is `null`, the result is `null`.

If the expression contains a method call, the rules are different. The period operator does not default to null if the left side of the period is null. With the standard period operator, if the value on the left of the period is null, Gosu throws a null pointer exception (`NullPointerException`). In other words, method calls are not null-safe because the method could have side effects.

Note: Technically speaking, property access from a dynamic `property get` function could generate side effects, but this is strongly discouraged. If a property accessor has any side effects, convert the accessor into a method.

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, Gosu throws an `NullPointerException`. 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.

However, remember that evaluation in the expression is left-to-right. If `claimant` is null, the expression throws an exception because Gosu cannot call a method on null.

Example 3

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 prints “Benefits decision not made yet”, even if `claim` or `claim.BenefitsStatusDcsn` is null:

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

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
```

Using null-safe paths means that Gosu automatically returns null if any of the following are null:

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

However, if the type of the P3 property is int or char or another primitive type, then the expression a.P1.P2.P3 is not null safe.

Primitive types (in contrast to object types, which are descendants 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 throws a null pointer exception in those conditions.

How the Null-Safe Period Operator Handles Null

In contrast to the standard period character operator, the null-safe period operator `?.` always returns `null` if the left side of the operator is `null`. This works both for accessing properties and for invoking methods. If the left side of the operator is `null`, Gosu does not evaluate the right side of the expression.

The null-safe operator is particularly important for invoking methods because for methods, the standard period operator throws an exception if the left side of the period is `null`.

For property access in Gosu, the standard period and the null-safe period operator are equivalent. The period operator is already null-safe.

Null-safe Default Operator

Sometimes you might need to return a different value based on whether some expression evaluates to `null`. The Gosu operator `?:` results in the value of the left-hand-side if it is non-null, avoiding evaluation of the right-hand-side. If the left-hand side expression is `null`, Gosu evaluates the right-hand-side and returns that result.

For example, suppose there is a variable `str` of type `String`. At run time the value contains either a `String` or `null`. Perhaps you want to pass the input to a display routine. However, if the value of `str` is `null`, you want to use a default value rather than `null`. Use the `?:` operator as follows:

```
var result = str ?: "(empty)" // return str, but if the value is null return a default string
```

Null-safe Indexing for Arrays, Lists, and Maps

For objects such as arrays and lists, you can access items by index number, such as `myArray[2]`. Similarly, with maps (`java.util.Map` objects), you can pass the key value to obtain the value. For example with a `Map<String, Integer>`, you could use the expression `myMap["myvalue"]`. The challenge with indexes is that if the object at run time has the value `null`, code like this throws a null pointer exception.

Gosu provides an alternative version of the indexing operator that is null-safe. Instead of simply typing the indexing subexpression, such as `[2]` after an object, prefix the expression with a question mark character. For example:

```
var v = myArray?[2]
```

If the value to the left of the question mark is `null`, the entire expression for the operator returns `null`. If the left-hand-operand is not `null`, Gosu looks inside the index subexpression and evaluates it and indexes the array, list or map. Finally, Gosu returns the result, just like the regular use of the angled brackets for indexing lists, arrays, and maps.

Null-safe Math Operators

Gosu provides null-safe versions of common math operators.

For example, the standard operators for addition, subtraction, multiplication, division, and modulo are as follows: `+`, `-`, `*`, `/`, and `%`. If you use these standard operators and either side of the operator is `null`, Gosu throws a `NullPointerException` exception.

In contrast, the null-safe operators are the same symbols but with a question mark (?) character preceding it. In other words, the null-safe operators are: `?+`, `?-`, `?*`, `?/`, and `?%`.

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” on page 87
- “Gosu Variables” on page 88
- “Gosu Conditional Execution and Looping” on page 92
- “Gosu Functions” on page 96

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 231.

Statement Lists

A statement list is a list containing zero or more Gosu statements beginning and ending with curly braces “{” and “}”, respectively.

It is the Gosu standard always to omit semicolon characters in Gosu at the end of lines. Code is more readable without optional semicolons. In the more rare cases in which you type multiple statement lists on one line, such as within block definitions, use semicolons to separate statements. For other style guidelines, see “General Coding Guidelines” on page 395.

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; }
```

New Is Optionally a Statement

Use the new operator to instantiate an object. See “New Object Expressions” on page 73. Although it is often used as an expression, it can also be a statement. For some types, this may not be useful.

However, if the constructor for the object triggers code that saves a copy of the new object, the return value from new may be unnecessary. Ignoring the return value and using new as a statement may permit more concise code in some cases.

For example, suppose that a constructor for a class that represents a book registers itself with a bookshelf object and saves the new object. Some code might simply create the book object and pass the bookshelf as a constructor argument.

```
new gw.example.Book( bookshelfReference, author, bookID )
```

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 previously 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.
*=	Multiples 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-pointing 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 logical OR 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. Both operators work with the primitive type boolean or the object type Boolean on either side of the operator	var a = false var b = true a = b Assigns value true.
&&=	Performs a logical 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. Both operators work with the primitive type boolean or the object type Boolean on either side of the operator	var a = false var b = true a &&= b Assigns value false.

Operator	Description	Examples
<code>&=</code>	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.	<pre>i = 10 i &= 15</pre> <p>Assigns value 10.</p> <p>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.</p> <p>Contrast with this example:</p> <pre>i = 10 i &= 13</pre> <p>Assigns value 8.</p> <p>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.</p>
<code>^=</code>	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.	<pre>i = 10 i ^= 15</pre> <p>Assigns value 5.</p> <p>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.</p> <p>Contrast with this example:</p> <pre>i = 10 i ^= 13</pre> <p>Assigns value 7.</p> <p>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.</p>
<code> =</code>	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.	<pre>i = 10 i = 15</pre> <p>Assigns value 15.</p> <p>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.</p> <p>Contrast with this example:</p> <pre>i = 10 i = 3</pre> <p>Assigns value 11.</p> <p>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.</p>
<code><<=</code>	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.	<pre>i = 10 i <<= 1</pre> <p>Assigns value 20.</p> <p>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.</p> <p>Contrast with this example:</p> <pre>i = 10 i <<= 2</pre> <p>Assigns value 40.</p> <p>The decimal number 10 is 001010 binary. This code does a bitwise left shift of 001010 one bit to the left. The result is binary 101000, which is decimal 40.</p>

Operator	Description	Examples
>>=	<p>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.</p> <p>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.</p>	<pre>i = 10 i >>= 1</pre> <p>Assigns value 5.</p> <p>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.</p> <p>Contrast with this example:</p> <pre>i = -10 i >>= 2</pre> <p>Assigns value -3.</p> <p>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.</p>
>>>=	<p>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.</p> <p>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.</p>	<pre>i = 10 i >>>= 1</pre> <p>Assigns value 5.</p> <p>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.</p> <p>Contrast with this example:</p> <pre>i = -10 i >>>= 2</pre> <p>Assigns value 1073741821.</p> <p>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.</p>
++ unary operator	<p>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</p> <p>IMPORTANT: See related information in “Compound Assignment Compared to Expressions” on page 91.</p>	<pre>i = 10 i++</pre> <p>Assigns value 11.</p>
-- unary operator	<p>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</p> <p>IMPORTANT: See related information in “Compound Assignment Compared to Expressions” on page 91.</p>	<pre>i = 10 i--</pre> <p>Assigns value 9.</p>

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 type's expression if the type is a subtype of the original type. This is particularly useful 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 357

For Statements

The `for` statement block uses a multi-part construction.

Syntax

```
for ( [var] <identifier> in <expression> [ index <identifier> ] ) { <statement> }
```

The scope of the `<identifier>` is limited to the statement block itself. The keyword `var` before the local variable identifier is optional.

The `<expression>` in the `in` clause must evaluate to one of the following:

- An array
- An iterator – an object that implements the Java `Iterable` interface. Iteration starts with the initial member and continues sequentially until terminating at the last member.
- A Java list or collection class, such as `java.util.ArrayList`. Iteration starts with the initial member and continues sequentially until terminating at the last member.
- A `String` object, which Gosu treats as a list of characters
- A Gosu interval – See “Intervals” on page 111.

See also “List Access Using Array Index Notation” on page 52 for details on using lists as arrays and accessing list members using array notation.

Gosu provides backwards compatibility for the use of the older style `foreach(...)` statement. However, it is best to use the `for(...)` statement instead.

Iteration in For Statements

There are several ways to iterate through members of a list or array using the `for` statement.

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.

Automatic Iteration With Variable

The standard form of iteration is to use an iteration variable. Use the following syntax:

```
for ( member in OBJ ) { ... }
```

Examples:

```
for( property in myListOfProperties )
    for( name in {"John", "Mary", "David"} )
        for( i in 0..|100 )
```

Automatic Iteration with No Variable - For Numerical Intervals Only

If the object to loop across is numerical iterator type, the variable declaration is optional. For example:

```
for (1..10) {
    print( "hello!" )
}
```

Automatic Iteration with Index

Use index iteration if you need to determine the exact position of a particular element within an array or list. This technique adds an explicit variable to contain the index value. This can be useful to read members of the array or list in a non-sequential fashion using array notation. Specify iteration with an index variable with 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 Example

Use this type of iteration if the object over which you are iterating is **not** a list or array, but it is an iterator.

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

```

See “Using the Results of Find Expressions (Using Query Objects)” on page 185 for additional iteration examples using the `iterator()` method.

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'.
for( var exp in Claim.Exposures ) {
    if( exp.OtherCoverage ) { // OtherCoverage is a Boolean property.
        print( "Found an exposure with other coverage." )
        // Transfer control to statement following this for...in statement
        break
    }
}

// Example 3: 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 `type is` 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 357.

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, declaring functions within a class. As in other object-oriented languages, *functions declared on a type* are also called *methods*.

In the context of a Gosu program (a `.gsp` file), you can declare functions at the top level, without attaching them explicitly to a class. You can then call this function from other places in that Gosu program.

Note: The built-in `print` function is special because it is always in scope, and is not attached to a type. It is the only true global function in Gosu.

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 231 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. Gosu 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  
}
```

IMPORTANT For more information about modifiers that can appear before the word `function` in class definitions, see “[Modifiers](#)” on page 198.

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, you can fix the earlier example 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.

Named Arguments and Argument Defaults

In code that calls functions, you can specify argument names explicitly rather than relying on matching the declaration order of the arguments. This helps make your code more readable. For example, typical method calls might look like the following:

```
someMethod(true, false) // what do those values represent? difficult to tell visually
```

Instead of passing simply a series of one or more comma-separated arguments, pass a colon, then the argument name, then the equals sign, then the value.

For example:

```
someMethod(:redisplay=true, :sendUpdate=false) // easy to read code!
```

Additionally, this feature lets you provide default argument values in function declarations. The function caller can omit that argument. If the function caller passes the argument, the passed-in value overrides any declared default value. To declare a default, follow the argument name with an equals sign and then the value.

To demonstrate default arguments, imagine a function that prints strings with a prefix:

```
class MyClass {  
    var _names : java.util.ArrayList<String>  
  
    construct( strings : java.util.ArrayList<String>) {  
        _strings = strings  
    }  
  
    function printWithPrefix( prefix : String = " ---> ") {  
        for( n in _strings ) {  
            print( prefix + n ) // used a passed-in argument, or use the default " ---> " if omitted  
        }  
    }  
}
```

Notice that in the `printWithPrefix` declaration, the prefix value has the default value " ---> ". To use the default values, call this class with the optional arguments omitted.

The following example shows calling the `printWithPrefix` method using the default and also a separate time overriding the default.

```
var c = new MyClass(["hello", "there"])  
  
// Because the argument has a default, it is optional -- you can omit it  
c.printWithPrefix()  
  
// Alternatively, specify the parameter to pass and override any default if one exists  
c.printWithPrefix(:prefix= " next string is:")
```

The Gosu named arguments feature requires that the method name is not already overloaded on the class.

Calling Conventions

When you call a function with a multiple arguments, you can name some of the arguments and not others. Any non-named arguments that you call must match in left-to-right order any arguments without defaults. Gosu considers any additional passed-in non-named arguments as representing the arguments with defaults, passed in the same order (left-to-right) as they are declared in the function.

If you use a named parameter in a function call, all following parameters must be named parameters.

Public and Private Functions

A function is public by default, meaning that it can be called from any Gosu code. 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()`.

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.

IMPORTANT See “Modifiers” on page 198 for more information on class and function level access modifiers.

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 as the using keyword.

This topic includes:

- “Try-Catch-Finally Constructions” on page 101
- “Throw Statements” on page 102
- “Catching Exceptions in Gosu” on page 103
- “Object Lifecycle Management (using Clauses)” on page 104
- “Assert Statements” on page 110

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.

Note: Gosu does not permit you to use a `return`, `break`, or `continue` statement in a `finally` block.

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 : Exception) {
    print( "Nested CATCH caught "+e )
    throw e + " rethrown"
  }
  finally { print( "Nested FINALLY running..." ) }
}
catch(e : Exception) { 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.

WARNING Do **not** use `throw` statements as part of regular (non-error) program flow. Use them only for handling actual error conditions.

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

```
uses java.lang.Exception

doOuterCode() // call outer code

function doOuterCode() {
  try {
    doInnerCode(0)
    doInnerCode(1)
  } catch(e : Exception) {
    print( e.Message + " -- caught in OUTER code" )
  }
}

function doInnerCode(x : int) {
  print("For value ${x}...")
```

```
try {
    if( x == 0 ) {
        throw "x equals zero"
    }
    else {
        throw "x does not equal zero"
    }
}
catch(e : Exception) {
    if( e.Message == "x equals zero" ) {
        print (e.message + " -- caught in INNER code.")
    }
    else { throw e }
}
```

This example prints:

```
For value 0...
x equals zero -- caught in INNER code.

For value 1...
x does not equal zero -- caught in OUTER code
```

Catching Exceptions in Gosu

Gosu allows you to catch and test for catch all general exceptions, or catch specific types of exceptions.

The standard syntax for catch is simply:

```
catch(e : Exception)
```

You can catch only specific exceptions by specifying a subclass such as `IOException` instead of `Exception`.

The following examples shows how this might look in practice:

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

IMPORTANT The recommended Gosu coding style is not to use checked exceptions. However, if you definitely need to handle a specific exception, use this concise syntax to make Gosu code more readable.

Add a `finally` block at the end to perform cleanup code that runs for errors and for success code paths:

```
try {
    doSomething()
}
catch( e : IOException ) {
}
finally {
    // PERFORM CLEANUP HERE
}
```

Throwable

The class `Throwable` is the superclass of all errors and exceptions in the Java language. However, it is best in general to use `catch(e : Exception)` not `catch(e : Throwable)`.

If you catch `Throwable`, it catches serious infrastructure problems like `OutOfMemoryException` or `AssertionFailedException`. In typical code, it is appropriate not to catch those exceptions. Let those throwable infrastructure exceptions propagate upward.

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 `using` statement. The `using` statement is a more compact syntax and less error-prone way to work with resources than using `try/catch/finally` clauses. With `using` clauses:

- Cleanup always occurs without requiring a separate `finally` clause.
- You do not need to explicitly check whether variables for initialized resources have `null` values.
- Code for locking and synchronizing resources is easy to use and understand.

For example, suppose you want to use an output stream. Typical code would open the stream, then use it, then close the stream 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 standard Java), you could 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 that code instead using the Gosu `using` statement:

```
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 list of one or more Gosu statements delimited by commas (not semicolons). For more details about using multiple initializable objects, see “Passing Multiple Items to the `using` Statement” on page 105

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.

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.

Return values from `using` clauses using the `return` statement, discussed further in “Returning Values from `using` Clauses” on page 108.

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 something that you want to reference from inside the `using` clause

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 a variable to 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 one or multiple items in the `using` clause expression. You must separate each item by a comma character, not a semicolon character as in a typical Gosu statement list.

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 104. Gosu runs any statements, including variable assignment, at run time and uses the result as an object to manage in the `using` clause. Within each comma-delimited assignment statement, you do not need a `return` statement.

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. Gosu cleans up only the objects that initialized without throwing exceptions or otherwise having errors such as returning `null` for a resource.

If you choose to initialize multiple resources and some but not all objects in the list successfully initialize:

1. Gosu skips initialization for any of the items not yet initialized, in other words the ones that appear later in the initialization list.
2. Gosu skips execution of the main part of the `using` clause.
3. Gosu cleans up exactly the set of objects that initialized without errors. In other words, Gosu releases, closes, or disposes the object, depending on the type of object.

For example, suppose you try to acquire three resources, and the first one succeeds but the second one fails. Gosu does not attempt to acquire the third resource. Then Gosu cleans up the one resource that did acquire successfully.

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:

- The object implements the Gosu interface `IDisposable`. This interface contains only a single method called `dispose`. This method takes no arguments. Always use a type that implements `IDisposable` if possible due to faster run time performance.
- The object 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.

The following example shows a basic disposable object:

```
// create a simple disposable class -- it implements IDisposable
class TestDispose implements IDisposable {
    construct() {
        print("LOG: created my object!")
    }
    override function dispose() {
        print("LOG: disposed of my object! Note that you must support multiple calls to dispose.")
    }
}
```

The following code tests this object

```
using (var d = new TestDispose()) {
    print("LOG: This is the body of the 'using' statement.")
}
```

This code prints:

```
LOG: created my object!
LOG: This is the body of the 'using' statement.
LOG: disposed of my object! Note that you must support multiple calls to dispose.
```

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. Use 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 `close` method.

To help ensure that resources clean up appropriately even under error conditions, you must design your `close` 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 package: `ReentrantLock`, `ReadWriteLock`, `Condition`.
- You cast the object to the Gosu interface `IMonitorLock`. You can cast **any** arbitrary object to `IMonitorLock`. It is useful to cast Java monitor locks to this Gosu interface. For more information on this concept, refer to: [http://en.wikipedia.org/wiki/Monitor_\(synchronization\)](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.

There is much more information about concurrency APIs in the section “Concurrency” on page 369.

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 108.

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 a `String` value defining the `ProfilerTag` name:

```
new ProfilerTag("EventPayloadXML")
```

To use the profiler tag, pass it to a `using` clause, as follows:

```

using( new ProfilerTag("EventPayloadXML") ) {

    // 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 163 in the *System Administration Guide*.

Returning Values from using Clauses

You can return values from within `using` 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:

- For disposable objects, Gosu calls the `dispose` method
- For closable objects, Gosu calls the `close` method
- For a reentrant or lock object, Gosu calls the `exit` method.

See previous topics for more information about each type of object you can use with a `using` clause.

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
}
```

Optional Use of a `finally` Clause with a `using` Clause

In some cases, you might need to perform additional cleanup that is not handled automatically by one of the closable, reentrant, or disposable types. To add arbitrary cleanup logic to the `using` clause, add a `finally` clause after the `using` clause. The Gosu `using` clause with the `finally` clause is similar to the `try-with-resources` statement of Java 1.7 but without a `catch` clause.

Important notes about the `finally` clause:

- The body of the `using` clause always runs before the `finally` clause.
- If exceptions occur that are not caught, Gosu still runs the `finally` clause.
- Your `finally` clause can access objects created in the `using` clause, but all resources are already cleaned up by the time the `finally` clause begins. For example, closeable objects are already closed and disposable objects are already disposed.

For example, create a disposable class:

```
// create a disposable class -- it implements IDisposable
class TestDispose implements IDisposable {

    var _open : boolean as OpenFlag

    construct(){
        print("LOG: created my object!")
        _open = true
    }
    override function dispose() {
        print("LOG: disposed of my object! Note that you must support multiple calls to dispose.")
        _open = false;
    }
}
```

The following code creates a disposable object in a `using` clause, and throws an exception in the body of the `using` clause. Note that the `finally` clause still runs:

```
using (var d = new TestDispose()) {
    print("LOG: This is the body of the 'using' statement.")
    print("value of d.OpenFlag = ${d.OpenFlag}")

    print(3 / 0) // THROW AN EXCEPTION (divide by zero)
}

finally {
    print("LOG: This is in the finally clause.")
    print("value of d.OpenFlag = ${d.OpenFlag}")
}

print("LOG: This is outside the 'using' clause.")
```

This example prints:

```
LOG: created my object!
LOG: This is the body of the 'using' statement.
```

```
value of d.OpenFlag = true  
java.lang.ArithmetricException: / by zero  
[...]  
LOG: disposed of my object! Note that you must support multiple calls to dispose.  
LOG: This is in the finally clause.  
value of d.OpenFlag = false
```

Notice in the output:

- The `using` clause automatically cleaned up the disposable resource.
- The `finally` clause still runs.
- The `finally` clause can access the resource, which is already disposed.

Assert Statements

An assert statement allows a concise syntax for asserting expectations and enforcing a programmatic contract with calling code. For this purpose, Gosu has an `assert` statement with the same semantics and syntax as in Java.

You can use the Gosu `assert` statement in two different forms

```
assert expressionBoolean  
assert expressionBoolean : expressionMessage
```

In the above syntax:

- `expressionBoolean` is a expression that returns a boolean result
- `expressionMessage` is an expression that returns a value that becomes a detailed message

For example:

```
assert i > 0 : i
```

By default, `assert` statements have no effect. Assertions are disabled.

To enable assertions, you must add the `-ea` flag on the JVM that hosts the application or Studio.

When assertions are enabled, Gosu evaluates the initial expression `expressionBoolean`:

- If the expression returns `true`, there is no effect from the `assert` statement.
- If the expression returns `false`, Gosu throws an `AssertionError` exception. If you use the version of the statement with a second expression, Gosu uses that value as an exception detail message. Otherwise, there is no exception detail message.

Intervals

An interval is a sequence of values of the same type between a given pair of endpoint values. Gosu provides native support for intervals. For instance, the set of integers beginning with 0 and ending with 5 is an integer interval containing the values 0, 1, 2, 3, 4, 5. The Gosu syntax for this is `0..5`. Intervals are particularly useful for concise easy-to-understand for loops. Intervals could be a variety of types including numbers, dates, dimensions, and names. You can add custom interval types. In other programming languages, intervals are sometimes called *ranges*.

What are Intervals?

An interval is a sequence of values of the same type between a given pair of endpoint values. Gosu provides native support for intervals. For instance, the set of integers beginning with 0 and ending with 10 is an integer interval. This interval contains the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. The Gosu syntax for this is `0..10`. Intervals are particularly useful for concise easy-to-understand for loops.

For example, consider this easy-to-read code:

```
for (i in 0..10) {  
    print("The value of i is " + i)  
}
```

This prints the following:

```
The value of i is 0  
The value of i is 1  
The value of i is 2  
The value of i is 3  
The value of i is 4  
The value of i is 5  
The value of i is 6  
The value of i is 7  
The value of i is 8  
The value of i is 9  
The value of i is 10
```

This replaces the more verbose and harder-to-read design pattern

```
var i = 0  
while( i <= 10 ) {  
    print("The value of i is " + i)  
    i++
```

```
}
```

Intervals do not need to be numbers. Intervals can be a variety of types including numbers, dates, dimensions, and names. Gosu includes built-in shorthand syntax with a double period for intervals for dates and common number types, such as the `0..10` example previously mentioned. The built-in shortcut works with the types `Integer`, `Long`, `BigInteger`, `BigDecimal`, and `Date`. All decimal types map to the `BigDecimal` interval.

You can also add custom interval types that support any type that supports iterable comparable sequences, and then you can use your new intervals in `for` loop declarations. For more information, see “Writing Your Own Interval Type” on page 113.

If you need to get a reference to the interval’s iterator object (`java.lang.Iterator`), call the `iterate` method and it returns the iterator.

Omitting an Initial or Ending Value

In the simple case, a Gosu interval iterates from the start endpoint to the ending endpoint and includes the values at both ends. For example, `0..5` represents the values `0, 1, 2, 3, 4, 5`.

In some cases, you want to exclude the beginning value but you still want your code to show the beginning value for code legibility. Similarly, some times you want to exclude the endpoint value from the interval.

To do this in Gosu, type the pipe “|” character:

- To make the starting endpoint open (to omit the value), type the pipe character after the starting endpoint.
- To make the ending endpoint open (to omit the value), type the pipe character before the ending endpoint.
- To do make both endpoints open, type the pipe character before and after the double period symbol.

Compare the following examples:

- `0..5` represents the values `0, 1, 2, 3, 4, 5`.
- `0|..5` represents the values `1, 2, 3, 4, 5`.
- `0...|5` represents the values `0, 1, 2, 3, 4`.
- `0|...|5` represents the values `1, 2, 3, 4`.

Reversing Interval Order

Sometimes you want a loop to iterate across elements in an interval in reverse order. To do this, reverse the position in relation to the double period symbol.

Compare the following examples:

- `0..5` represents the values `0, 1, 2, 3, 4, 5` (in that order).
- `5..0` represents the values `5, 4, 3, 2, 1, 0` (in that order).

Internally, they are the same objects but `5..0` is marked as being in reverse order.

For example, this iterates from 10 to 1, including the end points:

```
for( i in 10..1) {  
    print( i )  
}
```

If you have a reference to a reversed interval, you can force the interval to operate in its natural order. In other words, you can undo the flag that marks it as reversed. Use the following syntax:

```
var interv = 5..0  
var leftIterator = interv.iteratorFromLeft()
```

The result is that the `leftIterator` variable contains the interval for `0, 1, 2, 3, 4, 5`.

The `iterate` method, which returns the iterator, always iterates across the items in the declared order (either regular order or reverse, depending on how you defined it).

Granularity (Step and Unit)

You can customize the granularity with the `step` and `unit` builder-style methods on an interval. The `step` method lets you set the number of items to step (skip by). The `unit` method specifies the unit, which may or may not be necessary for some types of intervals. For example, the granularity of a date interval is expressed in units of time: days, weeks, months, hours. You could iterate across a date interval in 2 week periods, or 10 year periods, or 1 month periods.

Each method returns the interval so you can chain the result of one method with the next method. For example:

```
// Simple int interval visits odd elements
var interv = (1..10).step( 2 )

// Date interval visits two week periods
var span = (date1..date2).step( 2 ).unit( WEEKS )
```

Notice the `WEEKS` value. It is an enumeration constant and you do not need to qualify it with the enumeration type. Gosu can infer the enumeration type so the code is always type-safe.

Writing Your Own Interval Type

You can add custom interval types.

There are two basic types of intervals:

- Intervals you can iterate across, such as in `for` loop declarations. These are called *iterable intervals*.
- Non-iterable intervals

For typical code, intervals are the most useful if they are *iterable* because they can simplify common coding patterns with loops.

However, there are circumstances where you might want to create a non-iterable interval. For example, suppose you want to encapsulate an inclusive range of *real numbers* between two endpoints. Such a set includes a theoretically infinite set of numbers between the values 1 and 1.001. Iterating across the set is meaningless. For more information about creating non-iterable intervals, see “Custom Non-iterable Interval Types” on page 118.

The basic properties of an interval are as follows:

- The type of items in the interval must implement the Java interface `java.lang.Comparable`.
- The interval has left and right endpoints (the starting and ending values of the interval)
- Each endpoint can be closed (included) or open (excluded)

The main difference for iterable intervals is that they also implement the `java.lang.Iterable` interface.

Custom Iterable Intervals Using Sequenceable Items

The following example demonstrates creating a custom iterable interval using *sequenceable* items. A sequenceable item is a type that implements the `ISequenceable` interface. That interface defines how to get the next and previous items in a sequence. If the item you want to iterate across implements that interface, you can use the `SequenceableInterval` class (you do not need to create your own interval class). Suppose you want to create a new iterable interval that can iterate across a list of predefined (and ordered) color names with a starting and ending color value. Define an enumeration containing the possible color values in their interval:

```
package example.p1.gosu.interval

enum Color {
    Red, Orange, Yellow, Green, Blue, Indigo, Violet
}
```

For more information about creating enumerations, see “Enumerations” on page 209.

All Gosu enumerations automatically implement the `java.lang.Comparable` interface, which is a requirement for intervals. However, Gosu enumerations do not automatically implement the `ISequenceable` interface.

To determine an iterable interval dynamically, Gosu requires that a comparable endpoint also be *sequenceable*. To be sequenceable means that the class knows how to find the next and previous items in the sequence. Sequenceable and interval types have a lot in common. They both have the concept of granularity in terms of step amount and optionally a unit (such as weeks, months, and so on).

The interface for `ISequenceable` is as follows. Implement these methods and declare your class to implement this interface.

```
public interface ISequenceable<E extends ISequenceable<E, S, U>, S, U> {
    E nextInSequence( S step, U unit );
    E nextNthInSequence( S step, U unit, int iIndex );
    E previousInSequence( S step, U unit );
    E previousNthInSequence( S step, U unit, int iIndex );
}
```

The syntax for the interface might look unusual because of the use of Gosu generics. What it really means is that it is parameterized across three dimensions:

- The *type of each (sequenceable) element* in the interval.
- The *type of the step amount*. For example, to skip every other item, the step is 2, which is an `Integer`. For typical use cases, pass `Integer` as the type of the step amount.
- The *type of units* for the interval. For example, for an integer (1, 2, 3), choose `Integer`. For a date interval, the type is `DateUnit`. That type contains values representing days, weeks, or months. For instance, `DateUnit.DAYS`. If you do **not** use units with the interval, type `java.lang.Void` for this dimension of the parameterization. Carefully note the capitalization of this type, because it is particularly important to access Java types, especially when using Gosu generics. In Gosu, as in Java, `java.lang.Void` is the special type of the value `null`.

The example later in this topic has a class that extends the type:

```
IterableInterval<Color, Integer, void, ColorInterval>
```

For more information about Gosu generics, see “[Gosu Generics](#)” on page 239.

Notice that the interface can fetch both next and previous elements. It is bidirectional. Gosu needs this capability to handle navigation from either endpoint in an interval (the reverse mode). Gosu also requires the class know how to jump to an element by its index in the series. While this can be achieved with the single step methods, some sequenceable objects can optimize this method without having to visit all elements in between. For example, if the step value is 100, Gosu does not need to call the `nextInSequence` method 100 times to get the next value.

The following example defines an enumeration class with additional methods that implement the required methods of `ISequenceable`.

```
package example.pl.gs.int
uses java.lang.Integer

enum ColorSequencable
    implements gw.lang.reflect.interval.ISequenceable<ColorSequencable, Integer, java.lang.Void> {

    // enumeration values....
    Red, Orange, Yellow, Green, Blue, Indigo, Violet

    // required methods in ISequenceable interface...

    override function nextInSequence( stp : Integer, unit : java.lang.Void ) : ColorSequencable {
        return ColorSequencable.AllValues[this.Ordinal + stp]
    }

    override function nextNthInSequence( stp : Integer, unit : java.lang.Void,
        iIndex : int ) : ColorSequencable {
        return ColorSequencable.AllValues[this.Ordinal + stp * iIndex]
    }

    override function previousInSequence( stp : Integer, unit : java.lang.Void ) :
        ColorSequencable {
        return ColorSequencable.AllValues[this.Ordinal - stp]
    }
    override function previousNthInSequence( stp : Integer, unit : java.lang.Void,
        iIndex : int ) : ColorSequencable {
        return ColorSequencable.AllValues[this.Ordinal - stp * iIndex]
    }
}
```

```
    }  
}
```

To actually use this class, run the following code in Gosu Scratchpad:

```
print("Red to Blue as a closed interval...")  
var colorRange = new gw.lang.reflect.interval.SequenceableInterval(  
    ColorSequencable.Red, ColorSequencable.Blue, 1, null, true, true, false )  
  
for (i in colorRange) {  
    print(i)  
}  
  
print("Red to Blue as an open interval...")  
var colorRangeOpen = new gw.lang.reflect.interval.SequenceableInterval(  
    ColorSequencable.Red, ColorSequencable.Blue, 1, null, false, false, false )  
  
for (i in colorRangeOpen) {  
    print(i)  
}
```

This prints:

```
Red to Blue as a closed interval...  
Red  
Orange  
Yellow  
Green  
Blue  
Red to Blue as an open interval...  
Orange  
Yellow  
Green
```

If you wanted your code to look even more readable, you could create your own subclass of `SequenceableInterval` named for the sequenceable type you plan to use. For example, `ColorSequenceInterval`.

Custom Iterable Intervals Using Manually-written Iterators

If your items are not sequenceable, you can still make an iterable interval class but it takes more code to implement all necessary methods.

To create a custom iterable interval using manually-written iterator classes

1. Confirm that the type of items in your interval implement the Java interface `java.lang.Comparable`.
2. Create a new class that extends (is a subclass of) the `IterableInterval` class parameterized using Gosu generics across four separate dimensions:
 - The *type of each element* in the interval
 - The *type of the step amount*. For example, to skip every other item, the step is 2.
 - The *type of units* for the interval. For example, for an integer (1, 2, 3), choose `Integer`. For a date interval, the type is `DateUnit`. That type contains values representing days, weeks, or months. For instance, `DateUnit.DAYS`. If you do **not** use units with the interval, type `java.lang.Void` for this dimension of the parameterization. Carefully note the capitalization of this type, because it is particularly important to access Java types, especially when using Gosu generics. In Gosu, as in Java, `java.lang.Void` is the special type of the value `null`.
 - The *type of your custom interval*. This is self-referential because some of the methods return an instance of the interval type itself.

The example later in this topic has a class that extends the type:

```
IterableInterval<Color, Integer, void, ColorInterval>
```

For more information about Gosu generics, see “Gosu Generics” on page 239.

3. Implement the interface methods for the `Interval` interface.
4. Implement the interface methods for the `Iterable` interface.

The most complex methods to implement correctly are methods that return iterators. The easiest way to implement these methods is to define iterator classes as *inner classes* to your main class. For more information about inner classes, see “Inner Classes” on page 205.

Your class must be able to return two different types of iterators, one iterating forward (normally), and one iterating in reverse (backward). One way to do this is to implement a main iterator. Next, implement a class that extends your main iterator class, and which operates in reverse. On the class for the reverse iterator, to reverse the behavior you may need to override only the `hasNext` and `next` methods.

Example: Color Interval Written With Manual Iterators

In some cases, the item you want to iterate across does not implement the `ISequenceable` interface. You cannot modify it to directly implement this interface because it is a Java class from a third-party library. Although you cannot use the Gosu shortcuts discussed in “Custom Iterable Intervals Using Sequenceable Items” on page 113, you can still implement an iterable interval.

The following example demonstrates creating a custom iterable interval. Suppose you want to create a new iterable interval that can iterate across a list of predefined (and ordered) color names with a starting and ending color value. Define an enumeration containing the possible color values in their interval:

```
package example.pl.gosu.interval

enum Color {
    Red, Orange, Yellow, Green, Blue, Indigo, Violet
}
```

Note: For more information about creating enumerations, see “Enumerations” on page 209.

All Gosu enumerations automatically implement the `java.lang.Comparable` interface, which is a requirement for intervals.

Next, create a new class that extends the following type

```
IterableInterval<Color, Integer, void, ColorInterval>
```

Next, implement the methods from the `IIterableInterval` interface. It is important to note that in this example the iterator classes are inner classes of the main `ColorInterval` class.

```
package example.pl.gosu.interval
uses example.pl.gosu.interval.Color
uses gw.lang.reflect.interval.IterableInterval
uses java.lang.Integer
uses java.util.Iterator

class ColorInterval extends IterableInterval<Color, Integer, java.lang.Void, ColorInterval> {
    construct(left : Color, right : Color, stp : Integer) {
        super(left, right, stp)
        //print("new ColorInterval, with 2 constructor args")
    }

    construct(left : Color, right : Color, stp : Integer, leftOpen : boolean,
              rightOpen : boolean, rev: boolean) {
        super(left, right, stp, null, leftOpen, rightOpen, rev)
        //print("new ColorInterval, with 6 constructor args")
    }

    // get the Nth item from the beginning (left) endpoint
    override function getFromLeft(i: int) : Color {
        return Color.AllValues[LeftEndpoint.Ordinal + i]
    }

    // get the Nth item from the right endpoint
    override function getFromRight(i : int) : Color {
        return Color.AllValues[RightEndpoint.Ordinal - i]
    }

    // return standard iterator
    override function iterateFromLeft() : Iterator<Color> {
        var startAt = LeftEndpoint.Ordinal
        if (!LeftClosed)
            startAt++
        return new ColorIterator(startAt)
    }
}
```

```
// return reverse order iterator
override function iterateFromRight() : Iterator<Color> {
    var startAt = RightEndpoint.Ordinal
    if (!LeftClosed)
        startAt--
    return new ReverseColorIterator(startAt)
}

// DEFINE AN INNER CLASS TO ITERATE ACROSS COLORS -- NORMAL ORDER
class ColorIterator implements Iterator<Color>{
    protected var _currentIndex : int;

    construct() {
        throw "required start at # -- use other constructor"
    }

    construct(startAt : int ) {
        _currentIndex = startAt
    }

    override function hasNext() : boolean {
        return (_currentIndex) <= (RightEndpoint.Ordinal - (RightClosed ? 0 : 1)))
    }

    override function next() : Color {
        var i = _currentIndex
        _currentIndex++
        return Color.AllValues[i]
    }

    override function remove() {
        throw "does not support removing values"
    }
}

// DEFINE AN INNER CLASS TO ITERATE ACROSS COLORS -- REVERSE ORDER
class ReverseColorIterator extends ColorIterator {

    construct(startAt : int ) {
        super(startAt)
    }

    override function hasNext() : boolean {
        return (_currentIndex) >= (RightEndpoint.Ordinal + (LeftClosed ? 0 : 1)))
    }

    override function next() : Color {
        var i = _currentIndex
        _currentIndex--
        return Color.AllValues[i]
    }
}
```

Note the parameterized element type using Gosu generics syntax. It enforces the property that elements in the interval are mutually comparable.

Finally, you can use your new intervals in `for` loop declarations:

```
uses example.pl.gs.int.Color
uses example.pl.gs.int.ColorInterval

print("Red to Blue as a closed interval...")
var colorRange = new ColorInterval(
    Color.Red, Color.Blue, 1, true, true, false )

for (i in colorRange) {
    print(i)
}

print("Red to Blue as an open interval...")
var colorRangeOpen = new ColorInterval(
    Color.Red, Color.Blue, 1, false, false, false )

for (i in colorRangeOpen) {
    print(i)
}
```

This prints:

```
Red to Blue as a closed interval...
Red
Orange
Yellow
Green
Blue
Red to Blue as an open interval...
Orange
Yellow
Green
```

Custom Non-iterable Interval Types

There are circumstances where a range of numbers is non-iterable. For example, suppose you want to encapsulate an inclusive range of real numbers between two endpoints. Such a set would be inclusive to a theoretically infinite set of numbers even between the values 1 and 1.001. Iterating across the set is meaningless.

To create a non-iterable interval type, create a new class that descends from the class `AbstractInterval`, parameterized using Gosu generics on the class of the object across which it iterates. For example, to iterate across `MyClass` objects, mark your class to extend `AbstractInterval<MyClass>`.

The class to iterate across must implement the `Comparable` interface.

A non-iterable interval cannot be used in `for` loop declarations or other types of iteration.

Calling Java from 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 Gosu code that works with Java
- how to write Java code that works with Gosu and work with Guidewire products

For differences between the syntax of Gosu and Java as programming languages, see “Gosu Introduction” on page 15 and “Notable Differences Between Gosu and Java” on page 38.

To write your own Java to implement a plugin interface or to call your Java from Gosu, see:

- “Java and OSGi Support” on page 627 in the *Integration Guide*
- “Plugin Overview” on page 163 in the *Integration Guide*

This topic includes:

- “Overview of Writing Gosu Code that Calls Java” on page 119

Overview of Writing Gosu Code that Calls Java

Because Gosu is based on the Java Virtual Machine, Gosu code can directly use Java types. From Gosu, Java classes are first-class types and used like native Gosu classes. In most ways, Java syntax also works in Gosu. For example, instantiate a Java class with the new keyword:

```
var b = new java.lang.Boolean(false)
```

Gosu adds additional optional features and more concise syntax, such as optional type inference and optional semicolons. For differences between the syntax of Gosu and Java as programming languages, see “Gosu Introduction” on page 15 and “Notable Differences Between Gosu and Java” on page 38.

Gosu code can integrate with Java types in the following ways:

- instantiate Java types
- manipulate Java classes as native Gosu objects.
- manipulate Java objects as native Gosu objects.
- manipulate Java primitives as native Gosu objects. Gosu automatically converts between primitives and the equivalent object types automatically in most cases.
- get instance variables and static variables from Java types
- call methods on Java types. For methods that look like getters and setters, Gosu exposes methods also as *Gosu properties*. For more information, see “Java get/set/is Methods Convert to Gosu Properties” on page 121.
- add new methods to Java types using *Gosu enhancements*.
- add new properties to Java types using *Gosu enhancements*. (readable, writable, or read/write)
- add new properties to Java types automatically for methods that look like getters and setters. For more information, see “Java get/set/is Methods Convert to Gosu Properties” on page 121
- 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.

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. If your Java code needs to get, set, or query Guidewire entity data, you must understand how ClaimCenter works with entity data.

For more information, see “Java and OSGi Support” on page 627 in the *Integration Guide*.

Many Java Classes are Core Classes for Gosu

Because Gosu is built on the Java Virtual Machine. Many core Gosu classes are Java types. For example:

- the class `java.util.String` is the core 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

All types in the package `java.lang` are automatically in scope. Gosu code that uses types in that package does not need fully-qualified class names or explicit `uses` statements for those types.

For example, the following code is valid Gosu:

```
var f = new java.lang.Float(7.5)
```

However, the code is easier to understand with the simpler syntax:

```
var f = new Float(7.5)
```

For a larger list of types always in scope in Gosu, see “Importing Types and Package Namespaces” on page 79.

Static Members and Static Import 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 single type itself, not on *instances* of the type. 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 and later, which allows you to simply type PI instead of Math.PI.

This is only a syntax difference for using static members from Gosu, independent of whether the type is implemented in Gosu or Java). This does not affect how you write your Java code. For other syntax differences between Gosu and Java, see “Notable Differences Between Gosu and Java” on page 38.

Java get/set/is 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 as a property setter 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 as 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`.
- Gosu does this transformation on static methods as well as regular instance methods.
- Even though from Gosu there is a new property you can access for reading and writing this information, Gosu does not remove the methods. In other words, if a Java object has an `obj.getName()` method, you can use the expression `obj.Name` or `obj.getName()`.

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.

```

// instantiate 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. For example:

```
class MyGosuClass implements Runnable {  
    function run() {  
        // your code here  
    }  
}
```

Gosu also supports its own native interface definitions. For details, see “Interfaces” on page 211. 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 (descendants 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 362.

Java Generics

Gosu can directly access types that use Java generics. In typical code, it works identically.

Generics information is slightly different at run time if you get the run time type with the `typeof` operator or similar APIs:

- Due to how Java works, the parameterization of the type is erased by the Java compiler as it compiles to Java byte code. In other words, at run time, the run time type `ArrayList<Integer>` for a Java type appears just as `ArrayList`. This Java behavior is known as *type erasure*.
- In contrast, if the type is defined in Gosu, the run time type includes parameterization if present.

Query Builder APIs

The query builder APIs let you retrieve information from ClaimCenter application databases. The API framework mimics features of SQL SELECT statements to make your object-oriented Gosu queries easier to develop and optimize.

This topic includes:

- “Overview of the Query Builder APIs” on page 125
- “Building Simple Queries” on page 128
- “Joining Related Entities to Queries” on page 130
- “Restricting Queries with Predicates on Fields” on page 139
- “Working with Row Queries” on page 153
- “Working with Results” on page 158
- “Testing and Optimizing Queries” on page 171
- “Method and Type Reference for the Query Builder APIs” on page 176

Overview of the Query Builder APIs

The query builder APIs let you define and execute the equivalent of SQL SELECT statements against a ClaimCenter application database.

The Processing Cycle of Queries

Two types of objects drive the processing cycle of ClaimCenter queries:

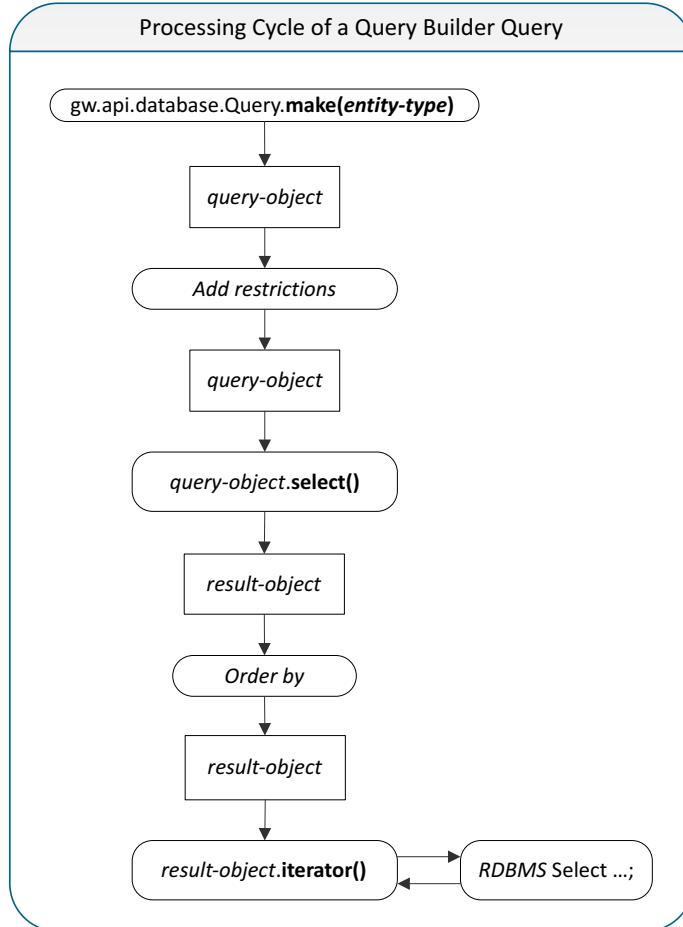
- **A query object** – Specifies which ClaimCenter entity instances to fetch from the application database
- **A result object** – A set of entity instances fetched from the database, based on the query object

The processing cycle of a ClaimCenter query follows these high-level steps.

1. Invoke the static `gw.api.database.Query.make(EntityType)` method, which creates a query object.
2. Refine your query object with restrictions.

3. Invoke the `select` method on your query object, which creates a result object.
4. Refine your result object by ordering the fetched items.
5. Iterate your result object with methods that the `java.lang.Iterable<T>` interface defines or with a `for` loop.

The following diagram illustrates the processing cycle of a query.



Note that the query builder APIs send queries to the application database when your code accesses information from the result set, not when your code calls the `select` method. Although your code seems to order results *after* fetching data, at runtime the application database orders the results *while* fetching data. Any action you take on result objects that returns information, such as getting result counts or starting to iterate the result, triggers execution of the query in the application database.

SQL Select Statements and Query Builder APIs Compared

Supported SQL Features in Query Builder APIs

Query builder APIs provide functionality similar to these features of SQL SELECT statements.

Keyword or clause in SQL	Equivalent in query builder APIs	Purpose
SELECT *	var query = Query.make(entity-type)	Begins a query. Generally, the results of a query are a set of object references to selected entity instances. For more information, see “Building Simple Queries” on page 128.
FROM <i>table</i>	var query = Query.make(entity-type)	Declares the primary source of data. For more information, see “Building Simple Queries” on page 128.
DISTINCT	query.withDistinct(true)	Eliminates duplicate items from the results. For more information, see “Avoiding Duplicate Instances” on page 136.
<i>column1 AS A[, column2 AS B ...]</i>	query.select(\row-> new HashMap<String, Integer>() { "A" -> row.field1[, "B" -> row.field2...] })	Produces a result that contains a set of name/value pairs, instead of a set of entity instances. For more information, see “Working with Row Queries” on page 153.
JOIN <i>table</i> ON <i>column</i>	var table = query.join("foreign-key")	Joins a secondary source of information to the primary source, based on a related column or field. For more information, see “Joining Related Entities to Queries” on page 130.
WHERE	query.compare("field-name", parameters) query.compareIgnoreCase("field-name", parameters) query.between("field-name", parameters) query.compareIn("field-name", parameters) query.compareNotIn("field-name", parameters) query.startsWith("field-name", parameters) query.contains("field-name", parameters)	Fetches information that meets specified criteria. For more information, see “Restricting Queries with Predicates on Fields” on page 139.
ORDER BY <i>column1[, column2[...]</i>	var orderedResult = result.orderBy(\row -> row.Field1)[.thenBy(\row -> row.Field2)[...]]	Sort results by specific columns or fields. For more information, see “Ordering Results” on page 165.
GROUP BY	Implied by aggregate functions on fields	Return results grouped by common values in a field. For an example query builder expression that generates an implied SQL GROUP BY clause, see “Database Aggregate Functions Within Select Blocks” on page 155.

Keyword or clause in SQL	Equivalent in query builder APIs	Purpose
HAVING	query.having()	Return results based on values from aggregate functions.
UNION	var union = query1.union(query2)	Combine items fetched by two separate queries into a single result.
TOP FIRST LIMIT	result.getCountLimitedBy(int)	Limit the results to the first <i>int</i> number of rows at the top, after grouping and ordering. For more information, see “Determining if a Result Will Return Too Many Items” on page 168.
INTERSECT	var intersection = query1.intersect(query2)	Reduce items fetched by two separate queries to those items in both results only.

Unsupported SQL Features in Query Builder APIs

Query builder APIs provide have no equivalent for these features of SQL SELECT statements. ClaimCenter never generates SQL statements that contain these keywords or clauses.

Keyword or clause in SQL	Meaning in SQL	Why the APIs do not support it
FROM <i>table1</i> , <i>table2</i> [, ...]	Declares the tables in a join query, beginning with the primary source of information on the left and proceeding with secondary sources of information towards the right.	<ul style="list-style-type: none"> This natural way of specifying a join can produce inappropriate results if the WHERE clause is not written correctly. Relational query performance often suffers when SQL queries include this syntax. You can specify only natural inner joins with this SQL syntax. <p>For an alternative with the query builder APIs, see “Joining Related Entities to Queries” on page 130.</p>
EXCEPT	Removes items fetched by a query that are in the results fetched by a second query.	This SQL feature is seldom used.

Building Simple Queries

Consider simple queries in SQL that return information from a single database table. The SQL SELECT statement specifies the table. Without further restrictions, the database returns all rows and columns of information from the table.

For example, you submit the following SQL statement to a relational database.

```
SELECT * FROM addresses;
```

In response, the relational database returns a *result set* that contains fetched information. A result set is like a database table, with columns and rows, that contains the information that you specified with the SELECT statement. In response to the preceding SQL example code, the relational database returns a result set that has the same columns and rows as the addresses table.

In comparison with the preceding SQL example, the following Gosu sample code constructs and executes a functionally equivalent query.

```
uses gw.api.database.Query      // Import the query builder APIs.

var query = Query.make(Address)
```

```
var result = query.select()
var iterator = result.iterator() // Execute the query and access the results with an iterator.
```

In response, the ClaimCenter application database returns a *result object* that contains fetched entity instances. A result object is like a Gosu collection that contains entity instances of the type that you specified with the `make` method and that meet the restrictions that you added. In response the `iterator` method in the preceding Gosu sample code, the application database fetches all Address instances from the application database into the result object.

Restricting the Results of a Simple Query

Generally, you want to restrict the information that queries return from a database instead of selecting all the information that the database contains. With SQL, the `WHERE` clause lets you specify Boolean expressions that data must satisfy to be included in the result.

For example, you submit the following SQL statement to a relational database.

```
SELECT * FROM addresses
WHERE city = "Chicago";
```

In response, the relational database returns a result set with addresses only in the city of Chicago. In the preceding SQL example, the Boolean expression applies the predicate `= "Chicago"` to the column `city`. The expression asserts that addresses in the result set have “Chicago” as the city.

In comparison with the preceding SQL example, the following Gosu sample code constructs and executes a functionally equivalent query.

```
uses gw.api.database.Query

var query = Query.make(Address)
query.compare("City", Equals, "Chicago")
var result = query.select()
var iterator = result.iterator()
```

In response to the preceding Gosu sample code, the application database fetches all Address instances from the application database that are in the city of Chicago.

Ordering the Results of a Simple Query

Relational databases return result sets with rows in a seemingly random order. With SQL, the `ORDER BY` clause lets you specify how the database sorts fetched data. The following SQL statement sorts the result set on the postal codes of the addresses.

```
SELECT * FROM addresses
WHERE city = "Chicago"
ORDER BY postal_code;
```

In comparison with the preceding SQL example, the following Gosu sample code sorts the instances fetched into the result object in the same way.

```
uses gw.api.database.Query

var query = Query.make(Address)
query.compare("City", Equals, "Chicago")
var result = query.select()
result.orderBy(\ row -> row.PostalCode) // Specify to sort the result by postal code.

var iterator = result.iterator() // Execute the query and return an iterator to access the result.
```

Accessing the Results of a Simple Query

Queries let you fetch information from a database and work with the results. Relational databases let you embed SQL queries in your application code so you can bind application data to SQL queries and submit them programmatically. Result sets returned by relational databases provide a programmatic cursor to let you access each result row, one after the other.

In comparison with SQL, the query builder APIs let you embed queries naturally in your Gosu application code. You bind application data to your queries and submit them programmatically with regular Gosu syntax. Result objects returned by ClaimCenter application databases implement the `java.lang.Iterable<T>` interface to let you access each entity instance in the result, one after the other.

For example, the following Gosu sample code creates a query of the Address entity instances in the application database. The query binds the constant "Chicago" to the `compare` predicate on the property `City`, which restricts the results.

```
uses gw.api.database.Query

// Specify what you want to select.
var query = Query.make(Address)
query.compare("City", Equals, "Chicago")

// Specify how you want the result returned.
var result = query.select()
result.orderBy(\ row -> row.PostalCode)

// Fetch the data with a for loop and print it.
for (address in result) {
    print (address.AddressLine1 + ", " + address.City + ", " + address.PostalCode)
}
```

When the preceding sample code begins the `for` loop, the application database fetches Address instances in the city of Chicago, sorts them by postal code. Then, one-by-one, the `for` loop passes addresses in the result to the statements inside the `for` block. The `for` block prints each address on a separate line.

Alternatively, the following Gosu sample code uses an iterator with a `while` loop to fetch matching Address instances in the order specified.

```
uses gw.api.database.Query

// Specify what you want to select.
var query = Query.make(Person)
query.compare("City", Equals, "Chicago")

// Specify how you want the result returned.
var result = query.select()
result.orderBy(\ row -> row.PostalCode)

// Fetch the data with a for loop and print it.
iterator = result.iterator()
while (iterator.hasNext()) {
    var address = iterator.next()
    print (address.AddressLine1 + ", " + address.City + ", " + address.PostalCode)
}
```

Note that there are no consequential functional or performance differences between iterating results with a `for` loop and iterating results with an iterator.

Joining Related Entities to Queries

To build useful queries in ClaimCenter, you often must join related entities to the primary entity of the query. The query builder APIs let you set restrictions only on database-backed fields, which store their values directly in the database. However, primary entities mostly have arrays, foreign keys, type lists, and derived fields. When you join related entities to a query, you can set restrictions on the related fields, which in turn restricts the primary entities in the result.

This topic contains:

- “Joining an Entity to a Query with a Simple Join” on page 131
- “Restricting Query Results with Fields on Joined Entities” on page 132
- “Different Ways to Join Related Entities to Queries” on page 133
- “Making a Query with an Inner Join” on page 133
- “Making a Query with a Left Outer Join” on page 136

- “Adding Predicates to Joined Entities” on page 137
- “Handling Duplicates in Joins with Foreign Keys on the Right” on page 138

Joining an Entity to a Query with a Simple Join

The term *join* comes from relational algebra and has special meaning for SQL and the query builder APIs.

Joining Tables in SQL Select Statements

In SQL, you can join two tables to form a new, virtual table, with rows from each table joined together based on columns with values that match. Then, you specify restrictions on rows in the join table, which have columns from both tables. When the query runs, the SQL result set has rows and columns from the join table.

The following example SQL Select statement uses the `JOIN` keyword to join the `addresses` table to the `companies` table.

```
SELECT * FROM companies
JOIN addresses
ON companies.primary_address = addresses.ID;
```

In response to the preceding example SQL statement, the database returns a result set with all the `companies` table that have a primary address. The result set includes columns for the companies, and it includes columns for their primary addresses. Companies without primary addresses are not included in the result.

For example, the `companies` and `addresses` tables have the following rows of information.

companies:		
id	name	primary_address
c:1	Hoffman Associates	a:1
c:2	Golden Arms Apartments	a:2
c:3	Industrial Wire and Chain	
c:4	North Creek Auto	a:3
c:5	Jamison & Sons	a:4

addresses:			
id	street_address	city	postal_code
a:1	123 Main St.	White Bluff	AB-2450
a:2	45112 E. Maplewood	Columbus	EF-6370
a:3	3945 12th Ave.	Arlington	IB-4434
a:4	930 Capital Way	Arlington	IR-8775

The preceding example SQL statement returns the following result set.

Result set with companies and addresses tables joined:

id	name	primary_address	id	street_address	city	postal_code
c:1	Hoffman Associates	a:1	a:1	123 Main St.	White Bluff	AB-2450
c:2	Golden Arms Apartments	a:2	a:2	45112 E. Maplewood	Columbus	EF-6370
c:4	North Creek Auto	a:3	a:3	3945 12th Ave.	Arlington	IB-4434
c:5	Jamison & Sons	a:4	a:4	930 Capital Way	Arlington	IR-8775

Note that the company named “Industrial Wire and Chain” is not in the result set. That row in the companies table has `null` as the value for `primary_address`, so no row in the addresses table matches. Note also that the columns in the result set are a union of the columns in the companies and addresses tables.

Joining Entities with the Query Builder APIs

With the query builder APIs, you can join a related entity type to the primary entity type of the query. When you join an entity type to a query, the query builder APIs construct an internal object to represent the joined entity and how it participates in the query.

In the simplest case, use the `join` method to join a secondary entity type to a query. You must specify the name of a property on the primary entity that the *Data Dictionary* defines as a foreign key to the secondary entity. Use the Gosu property name, not the actual column name in the database.

For example, to join the Address entity to a query of the Company entity, use the `PrimaryAddress` property of Company, which is a foreign key to Address.

```
uses gw.api.database.Query

var query = Query.make(Company)
query.join("PrimaryAddress") // Join the Address entity to the query by a foreign key on Company.
var result = query.select()

// Fetch the data with a for loop and print it.
for (company in result) {
    print (company)
}
```

Using the sample data shown earlier, the preceding query builder API query produces the following result.

Result object with Company instances from a join query:

Name

Hoffman Associates
Golden Arms Apartments
North Creek Auto
Jamison & Sons

Note that the Company named “Industrial Wire and Chain” is not in the result object. This instance has `null` as its `primary_address`, so no instance of Address matches.

Note also that whereas SQL returned information from both tables, the query builder APIs return only instances of the primary entity type. With the query builder APIs however, you can use dot notation to access information from the joined entity type. For example:

```
uses gw.api.database.Query

var query = Query.make(Company)
query.join("PrimaryAddress") // Join the Address entity to the query by a foreign key on Company
var result = query.select()

// Fetch the data with a for loop and print it --
for (company in result) {
    print (company + ", " + company.PrimaryAddress.City)
}
```

Restricting Query Results with Fields on Joined Entities

When you join entity types to a query, the `join` method creates an internal object that represents the joined entity. The method returns an object reference to it. If you capture the object reference in a variable, you can perform other operations on the joined entity that affect how the query selects items for the result.

For example, you want a query that returns all companies in the city of Chicago. The primary entity type of your query is `Company`, because you want instances of that type in your result.

```
var queryCompany = Query.make(Company)
```

The `Company` entity does not have a `City` property. That property is on the `Address` entity. So, you need to join the `Address` entity to your query. You must capture the object reference that the `join` method returns so you can specify predicates on values in `Address`.

```
var tableAddress = queryCompany.join("PrimaryAddress")
tableAddress.compare("City", Equals, "Chicago")
```

When you run the query, the result contains companies that have Chicago as the city on their primary addresses.

```
uses gw.api.database.Query

// Start a new query with Company as the primary entity.
var queryCompany = Query.make(Company)

// Join Address as the secondary entity to the primary entity Company.
var tableAddress = queryCompany.join("PrimaryAddress")

// Add a predicate on the secondary entity.
tableAddress.compare("City", Equals, "Chicago")

// Fetch the data with a for loop.
var result = queryCompany.select()
for (company in result) {
    print(company + ", " + company.PrimaryAddress.City)
}
```

Different Ways to Join Related Entities to Queries

In SQL, you can join a table to a query in these ways:

- **Inner join** – Excludes rows/instances on the left that have no matches on the right
- **Outer left join** – Includes rows/instances from the left that have no matches on the right
- **Outer right join** – Include rows/instances from the right that have no matches on the left
- **Full outer join** – Include rows/instances that have no matches, regardless of side

The query builder APIs support inner joins and outer left joins only. Outer right joins and full outer joins *are not* supported.

For more information, see:

- “Making a Query with an Inner Join” on page 133
- “Making a Query with a Left Outer Join” on page 136

Making a Query with an Inner Join

With an inner join, items from the primary table or entity on the left are joined to items from the table or entity on the right, based on matching values. If a primary item on the left has no matching value on the right, that primary item is not included in the result. A foreign key from one side to the other is the basis of matching.

In SQL, it generally does not matter which side of the join the foreign key is on. It generally does not matter whether the foreign key is defined in metadata. It does not matter if the column or property names are the same. All that matters is for one side of the join to have a column or property with values that match values in the column of another table or property.

With the query builder APIs, it matters which side of a join the foreign key is on. The side that has the foreign key determines which `join` method signature to use.

- `join("column")` – Joins two entities when the foreign key is on left
See “Making Inner Joins with the Foreign Key on the Left” on page 134.
- `join(table, "column")` – Joins two entities when the foreign key is on the right.
See “Making Inner Joins with the Foreign Key on the Right” on page 135.

Making Inner Joins with the Foreign Key on the Left

In SQL, you make an inner join with the `JOIN` keyword. You specify the table that you want to join to the primary table of the query. The `ON` keyword lets you specify which columns join the tables. In the following SQL statement, the foreign key, `primary_address`, is on the primary table, `companies`. So, the foreign key is on the left side.

```
SELECT * FROM companies
  JOIN addresses
    ON companies.primary_address = addresses.ID;
```

With the query builder APIs, use the `join` method with a single parameter if the primary entity has the foreign key. In these cases, the foreign key is on the left. You specify a property of the primary entity that the *Data Dictionary* defines as a foreign key to the secondary entity. The query builder APIs use metadata to determine the entity type to which the foreign key relates.

Joining the Secondary Entity to a Query with the Foreign Key on the Left

In the following Gosu sample code, the primary entity type, `Company`, has a foreign key, `PrimaryAddress`, which relates to the secondary entity type, `Address`.

```
var queryCompany = Query.make(Company)
queryCompany.join("PrimaryAddress")
```

Notice that unlike SQL, you do not specify which secondary entity to join, in this case `Address`. Neither do you specify the property on the secondary entity, `ID`. The query builder APIs use metadata from the *Data Dictionary* to fill in the missing information.

Applying Predicates to the Secondary Entity

With the query builder APIs, generally you join a secondary entity to a query only so you can restrict the results with predicates on the secondary entity. Unlike SQL, the result of a query contains entity instances, not table rows with columns. You can access any information related to the primary entity in the result with dot notation, regardless of including the related entity type in the query itself.

The following Gosu sample codes shows how to use dot notation to access information related to the primary entity of a query. The code accesses related `City` information from the `Address` entity with dot notation on `Company` instances retrieved from the result.

```
uses gw.api.database.Query

// -- query the Company entity --
var query = Query.make(Company)
var result = query.select()

// -- fetch the Company instances with a for loop and print them --
for (company in result) {
    print (company.Name + ", " + company.PrimaryAddress.City)
}
```

However, you seldom want to retrieve all instances of the primary type of a query. To select a subset, you often join a secondary entity and apply predicates to one or more of its properties. To apply predicates to the secondary entity in a join, you can save the `table` object that the `join` method returns in a local variable. Then, you can use predicate methods on the `table` object, such as `compare`, to restrict which secondary instances to include in the result. That also restricts which primary entities instances the result returns.

For example, you want to select only companies in the city of Chicago. So, you must join the `Address` entity to your `Company` query and apply the predicate = "Chicago" to it.

```
uses gw.api.database.Query

// -- query the Company entity --
var queryCompany = Query.make(Company)

// -- select only Company instances in the city of Chicago --
var tableAddress = queryCompany.join("PrimaryAddress")
tableAddress.compare("City", Equals, "Chicago")

// -- fetch the Company instances with a for loop and print them --
```

```

var result = queryCompany.select()
for (company in result) {
    print (company.Name + ", " + company.PrimaryAddress.City)
}

```

Note that the query result contains only Company instances. Even though the code above included the Address entity in the query, information from joined entities generally is not included in results. The code above uses dot notation to access information from related Address instances after retrieving Company instances from the result. By joining Address to the query and applying the predicate, the code ensures that Company instances in the result have only "Chicago" in the property company.PrimaryAddress.City.

Making Inner Joins with the Foreign Key on the Right

In SQL, you make an inner join with the JOIN keyword. You specify the table that you want to join to the primary table of the query. The ON keyword lets you specify which columns join the tables. In the following SQL statement, the foreign key, update-user, is on the secondary table, addresses. So, the foreign key is on the right side. The query returns a result with all addresses and users who lasted updated them.

```

SELECT * FROM users
JOIN addresses
ON users.ID = addresses.update_users;

```

With the query builder APIs, use the join method with two parameters if the secondary entity has the foreign key. In these cases, the foreign key is on the right. You specify the name of the secondary entity, followed the property that is a foreign key to the primary entity.

Joining the Secondary Entity to a Query with the Foreign Key on the Right

In the following Gosu sample code, the secondary entity type, Address, has a foreign key, UpdateUser, which relates to the primary entity type, User.

```

var query = Query.make(User)
query.join(Address, "UpdateUser")

```

Notice that just like SQL, you must specify which secondary entity to join, in this case Address. However unlike SQL, you do not specify the property on the primary entity, ID. The query builder APIs use metadata from the *Data Dictionary* to fill in the missing information.

When the foreign key is on the right, the data model definition of the column must specify a kind of link: a foreignkey, a onetoone, or an edgeForeignKey.

Applying Predicates to the Secondary Entity

With the query builder APIs, generally you join a secondary entity to a query only so you can restrict the results with predicates on the secondary entity. Unlike SQL, the result of a query contains entity instances, not table rows with columns. You can access any information related to the primary entity in the result with dot notation, regardless of including the related entity type in the query itself.

However, you seldom want to retrieve all instances of the primary type of a query. To select a subset, you often join a secondary entity and apply predicates to one or more of its properties. To apply predicates to the secondary entity in a join, you can save the table object that the join method returns in a local variable. Then, you can use predicate methods on the table object, such as compare, to restrict which secondary instances to include in the result. That also restricts which primary entities instances the result returns.

For example, you want to select only users who have updated addresses in the city of Chicago. So, you must join the Address entity to your User query, Then, you can and apply the predicate = "Chicago" to Address, the secondary entity.

```

uses gw.api.database.Query

// -- query the User entity --
var queryUser = Query.make(User)

// -- select only User instances who last updated addresses in the city of Chicago --
var tableAddress = queryUser.join(Address, "UpdateUser")

```

```

tableAddress.compare("City", Equals, "Chicago")

// -- fetch the User instances with a for loop and print them --
var result = queryUser.select()

for (user in result) {
    print (user.DisplayName)
}

```

Accessing Properties of the Secondary Entity

Note that the query result contains only `User` instances. Even though the code above included the `Address` entity in the query, information from joined entities generally is not included in results. This can be a problem with inner joins when the foreign key is on the right. It means that you cannot traverse from primary instances on the left to related instances on the right with dot notation.

In the preceding Gosu example query, you cannot determine which addresses specific users in the result last updated. In some cases this is not a problem. Often, you use the results of inner queries with foreign keys on the right for further processing without accessing *any* information from the related instances that led to the result. If you *do* want to access the information, for example to display on a screen or in a report, see “Working with Row Queries” on page 153.

Avoiding Duplicate Instances

Note that the query results probably contain main duplicate instances of the same user. When the foreign key is on the right side of a join, the foreign key value for any specific primary instance is likely to recur. This problem occurs with SQL and the query builder APIs alike.

Consider the preceding query, which selected those users who have updated addresses from the city of Chicago. The same user is likely to have updated several Chicago addresses. Therefore, the result contains that user once for every Chicago address updated. The following Gosu example code shows how to use the `withDistinct` method on the query.

```

uses gw.api.database.Query

// -- query the User entity --
var queryUser = Query.make(User)
queryUser.withDistinct(true)

// -- select only User instances who last updated addresses in the city of Chicago --
var tableAddress = queryUser.join(Address, "UpdateUser")
tableAddress.compare("City", Equals, "Chicago")

// -- fetch the User instances with a for loop and print them --
var result = queryUser.select()

for (user in result) {
    print (user.DisplayName)
}

```

For more information, see “Handling Duplicates in Joins with Foreign Keys on the Right” on page 138.

Making a Query with a Left Outer Join

With a left outer join, items from the primary entity on the left are joined to items from the entity on the right, based on matching values. If a primary item on the left has no matching value on the right, that primary item is included in the result, anyway. A foreign key from one side or the other is the basis of matching.

In SQL, it generally does not matter which side of the join the foreign key is on. It generally does not matter whether the foreign key is defined in metadata. It does not matter if the column or property names are the same. All that matters is for one side of the join to have a column or property with values that match values in the column of another table or property.

With the query builder APIs, it matters which side of a join the foreign key is on. The side that has the foreign key determines which `outerJoin` method signature to use:

- `outerJoin("column")` – Joins two entities when the foreign key is on left.

- `outerJoin(table, "column")` – Joins two entities when the foreign key is on the right.

Making Left Outer Joins with the Foreign Key on the Left

In SQL, you make a left outer join with the `LEFT JOIN` keywords. You specify the table that you want to join to the primary table of the query. The `ON` keyword lets you specify which columns join the tables. In the following SQL statement, the foreign key, `supervisor`, is on the primary table, `groups`. So, the foreign key is on the left side.

```
SELECT * FROM groups
  LEFT JOIN users
    ON groups.supervisor = users.ID;
```

With the query builder APIs, use the `outerJoin` method with a single parameter if the primary entity has the foreign key. In these cases, the foreign key is on the left. You specify a property of the primary entity that the *Data Dictionary* defines as a foreign key to the secondary entity. The query builder APIs use metadata to determine the entity type to which the foreign key relates.

Joining the Secondary Entity to a Query with the Foreign Key on the Left

In the following Gosu sample code, the primary entity type, `Group`, has a foreign key, `Supervisor`, which relates to the secondary entity type, `User`.

```
var queryGroup = Query.make(Group)
queryGroup.outerJoin("Supervisor") // Supervisor is a foreign key from Group to User.
```

Notice that unlike SQL, you do not specify which secondary entity to join, in this case `User`. Neither do you specify the property on the secondary entity, `ID`. The query builder APIs use metadata from the *Data Dictionary* to fill in the missing information.

Applying Predicates to the Secondary Entity

With the query builder APIs, generally you join a secondary entity to a query only so you can restrict the results with predicates on the secondary entity. Unlike SQL, the result of a query contains entity instances, not table rows with columns. You can access any information related to the primary entity in the result with dot notation, regardless of including the related entity type in the query itself.

The following Gosu sample codes shows how to use dot notation to access information related to the primary entity of a query. The code accesses the `DisplayName` information for the group supervisor from the `User` entity with dot notation on `Group` instances retrieved from the result.

```
uses gw.api.database.Query

// -- query the Group entity --
var queryGroup = Query.make(Group)
queryGroup.outerJoin("Supervisor") // Supervisor is a foreign key from Group to User.

// -- fetch the Group instances with a for loop and print them --
var result = queryGroup.select()

for (group in result) {
    if (group.Supervisor != null) //-- access the supervisor in a typesafe way --
        print (group.Name + ": " + group.Supervisor.Contact.DisplayName)
    else
        print (group.Name + ": " + "group has no supervisor")
}
```

Adding Predicates to Joined Entities

You can add predicates on the primary entity of a query after you make a new one. For example:

```
var query = Query.make(SampleParent)

query.compareIn("ID", myEntityIDs)
query.join("MyChildObjectProperty")
```

You can add predicates on a joined entity by calling predicate methods on the joined table. The values returned by the join methods are query builder table objects. Save the table object that a join method returns and call predicate methods on it, as the following sample Gosu codes shows.

```
var query = Query.make(SampleParent)
var joinTable = query.join("MyChildObjectProperty")
joinTable.compare("ChildPriority", Equals, 5)
```

Alternatively, you can chain the join method and call the predicate method on the returned table object in a single statement, as the following sample Gosu code shows.

```
var query = Query.make(SampleParent)
query.join("MyChildObjectProperty").compare("ChildPriority", Equals, 5)
```

You can add predicates to the primary entity even after you join another entity to query.

```
var query = Query.make(SampleParent)

var joinTable = query.join("MyChildObjectProperty")

// Add predicates to the join table by specifying fields of the joined entity.
joinTable.compare("ChildPriority", Equals, 5)

// Add predicates to the primary table by specifying fields of the primary entity.
query.compareIn("ID", myEntityIDs)
```

Handling Duplicates in Joins with Foreign Keys on the Right

It is easy to fetch duplicate instances of the primary entity when you use join or outerJoin with the foreign key on the right of the join.

```
var queryParent = Query.make(SampleParent)
var tableChild = queryParent.join(SampleChild, "Parent")
```

Using the previous example, if more than one SampleChild relates to the same SampleParent, that SampleParent is returned more than once. One instance exists in the result of the join query for each child object of type SampleChild that relates to that instance.

Duplicate instances of the primary entity in a query result 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.

Note: Using join and outerJoin with foreign keys on the right can return duplicate instances of the primary entity. This occurs if a primary instance on the left relates to multiple secondary instances on the right. To eliminate duplicates in a result, use the withDistinct method on your query.

Notice that duplicates can occur only with joins where the foreign key is on the right. When the foreign key for a join is on the left, the value for any primary instance is unique in the secondary table. So a primary instance on the left relates either to:

- A value of null, which matches no instance on the right
- A single instance on the right

For joins with the key on the right, 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. However, every situation is different, and in some cases, using the join method might perform better. For example, if the secondary entity includes predicates that 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 with the foreign key on the right, use the two-parameter method signature that includes the table name followed by the 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 query = Query.make(SampleParent)
query.withDistinct(true)
query.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 a related 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 query = Query.make(SampleParent)
query.join("MyChildObjectProperty")
query.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 query = Query.make(SampleParent)
query.join("MyChildObjectProperty")
query.having().compare("ChildPriority", Equals, 5)
```

The addition of the `having` method adds an 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. You can use this approach only if you add predicates to the join table.

Restricting Queries with Predicates on Fields

Predicates are conditions that select a subset of values from a larger set. You can apply predicates to different sets of values to select different subsets of values. Predicates are independent of the sets to which you apply them.

For example, the expression “is male” is a predicate. It comprises a comparison operator, “is,” and a comparison value, “male.” You can apply this predicate to different sets of people. If you apply the predicate “is male” to the set of people in your family, the predicate selects all the male members of your family. If you apply the predicate to the set of people in your work group, “is male” selects a different, possibly overlapping subset of males.

SQL Select statements provide `WHERE` and `HAVING` clauses to apply predicates to the sets of values in specific database columns. The query builder APIs provide functions on query objects to apply predicates to the sets of values in specific entity fields.

This topic contains:

- “Using Comparison Predicates with Character Fields” on page 140
- “Using Comparison Predicates with Date and Time Fields” on page 141
- “Using Comparison Predicates with Null Values” on page 143
- “Using Set Inclusion and Exclusion Predicates” on page 144
- “Comparing Column Values with Each Other” on page 145
- “Comparing Column Values with Literal Values” on page 145
- “Comparing Typekey Column Values with Typekey Literals” on page 146
- “Combining Predicates with AND and OR Logic” on page 147
- “Predicate Methods Reference” on page 151

Using Comparison Predicates with Character Fields

With SQL, you can apply comparison predicates to character fields by using Gosu character literals as comparison values. In SQL, you apply predicates to column values in the WHERE clause. For example, the following SQL statement applies the predicate = "Acme Rentals" to values in the name column of the companies table to select the company "Acme Rentals."

```
SELECT * from companies
WHERE name = "Acme Rentals";
```

With the query builder APIs, you apply predicates to entity fields by using functions on the query object. The following Gosu sample code applies the function compare to values in the Name field of Company entity instances to select the company "Acme Rentals."

```
uses gw.api.database.Query

// Query the Company instances for a specific company.
var query = Query.make(Company)
query.compare("Name", Equals, "Acme Rentals")

// Fetch the data print it.
var result = query.select()
for (company in result) {
    print (company.Name)
}
```

With comparison predicates applied to character fields, the values in the field must match exactly the comparison value, including the case of the comparison value. For example, the preceding query builder code selects only "Acme Rentals," not "ACME Rentals."

Note: ClaimCenter converts character literals that you specify in query builder predicates to bind variables in the generated SQL statement that ClaimCenter sends to the relational database.

See also

- “Comparing Column Values with Literal Values” on page 145
- “Comparing Typekey Column Values with Typekey Literals” on page 146

Case-Insensitive Comparisons with Character Fields

A problem often occurs when you compare character values. You might not know which case the character values that you want to select use. For example, you want to select the company named "Acme Rentals." There is one instance only of the Company entity type with that name. However, you do not know whether the company is in the database as "Acme Rentals," "ACME RENTALS," or even "acme rentals."

SQL Select statements let you specify case-insensitive comparisons with functions that convert values in columns to upper or lower case before making comparisons. The following example SQL statement converts values in the name column to lower case before applying the predicate = "acme rentals".

```
SELECT * from companies
WHERE LCASE(name) = "acme rentals";
```

The query builder APIs provide the compareEqualsIgnoreCase method for case-insensitive comparisons, as the following Gosu sample code shows.

```
uses gw.api.database.Query

var query = Query.make(Company)
query.compareIgnoreCase("Name", Equals, "Acme Rentals")

// Fetch the data print it.
var result = query.select()
for (company in result) {
```

```
    print (company.Name)
}
```

IMPORTANT Your data model definition must specify the column with the attribute `supportsLinguisticSearch` set to `true`. Otherwise, query performance suffers if you include the column in a `compareIgnoreCase` predicate method.

Partial Comparison Predicates with Character Fields

Sometimes you want to make partial comparisons that match the beginning or portions in the middle of character fields, instead of matching completely the entire field. SQL SELECT statements support the following wildcard characters for use with the LIKE operator:

- **% (percent sign)** – represents zero, one, or more characters
- **_ (underscore)** – represents exactly one character

The query builder APIs offer the following methods to simulate the SQL LIKE operator:

- `startsWith` – accepts a string of characters to match at the beginning of a character field
- `contains` – accepts a string of characters to match any portion of a character field

Case-insensitive Partial Comparisons

Sometimes you want to make partial comparisons in a case-insensitive way. SQL SELECT statements let you specify case-insensitive comparisons with functions that convert the values in columns to upper or lower case before making the comparison. The following SQL statement converts the values in `AddressLine1` to lower case before comparing them to `main st`.

```
SELECT * from companies a, addresses b
WHERE a.primary_address = b.ID
AND LCASE(b.AddressLine1) LIKE "%main st%";
```

In comparison with the preceding SQL example, the following Gosu sample code constructs and executes a functionally equivalent query.

```
uses gw.api.database.Query

// -- start a new query with Company as the primary entity --
var queryCompany = Query.make(Company)

// -- join Address as the secondary entity to the primary entity Company --
var tableAddress = queryCompany.join("PrimaryAddress")

// -- add a predicate on the secondary entity --
tableAddress.contains("AddressLine1", "Market St", true) // -- Equivalent to SQL LIKE.
//           // The true value means "case-insensitive"

// -- fetch the data with a for loop --
var result = queryCompany.select()
for (company in result) {
    print (company + ", " + company.PrimaryAddress.AddressLine1)
}
```

If you choose case sensitivity, Gosu generates an SQL LOWER function to implement the comparison predicate. However, if the data model definition of the column sets the `supportsLinguisticSearch` attribute to `true`, Gosu uses the denormalized version of the column instead.

Using Comparison Predicates with Date and Time Fields

Different brands of database have different functions that work with timestamp columns in SQL query statements. In contrast, the query builder APIs offer these database functions for working with timestamp columns:

- `DateDiff` – See “Comparing the Interval Between Two Date and Time Fields” on page 142
- `DatePart` – See “Comparing Parts of a Date and Time Field” on page 142
- `DateFromTimestamp` – See “Comparing the Date Part of a Date and Time Field” on page 143

Comparing the Interval Between Two Date and Time Fields

Use the static `DateDiff` method of the `DBFunction` class to compute the interval between two `DateTime` fields. The `DateDiff` method takes two `ColumnRef` parameters to timestamp fields in the database: a starting timestamp and an ending timestamp. It returns the interval between the two.

```
DateDiff(dateDiffPart : DateDiffPart, startDate : ColumnRef, endDate : ColumnRef) : DBFunction
```

The method has an initial parameter, `dateDiffPart`, that lets you specify the unit of measure for the result. You can specify `DAYS`, `HOURS`, `SECONDS`, or `MILLISECONDS`. If `endDate` precedes the `startDate`, the method returns a negative value for the interval, instead of a positive value.

Example of the `DateDiff` Method

The following sample Gosu code uses the `DateDiff` method to compute the interval between the assigned date and the due date on an activity. The query builder code uses the returned interval in a comparison predicate to select activities with due dates less than 15 days from their assignment dates.

```
uses gw.api.database.Query
uses gw.api.database.DBFunction

var query = Query.make(Activity)
// Query for activities with due dates less than 15 days from the assigned date.
query.compare(DBFunction.DateDiff(DAYS, query.getColumnRef("AssignmentDate"),
query.getColumnRef("EndDate")), LessThan, 15)

// Order the result by assignment date.
for (activity in (query.select().orderBy(\ row -> row.AssignmentDate))) {
    print("Assigned on " + activity.AssignmentDate + ": " + activity.DisplayName)
}
```

DateDiff Method Daylight Saving or Summer Time

The `DateDiff` method does not adjust for daylight saving, or summer, time. For example, daylight saving or summer time in your locale ends on the first Sunday in November. The `DateDiff` method operates on a row with the `DateTime` values `2011-11-5 12:00` and `2011-11-6 12:00`. The method returns an interval of 24 hours, even though 25 hours separate the two in solar time.

Comparing Parts of a Date and Time Field

Use the static `DatePart` method of the `DBFunction` class to extract a portion of a `DateTime` field to use in a comparison predicate. The `DatePart` method takes two parameters. The first parameter specifies the part of the date and time you want to extract, and the second parameter specifies the field from which to extract the part. It returns the extracted part as a `Number`.

```
DateDiff(datePart : DatePart, date : ColumnRef) : DBFunction
```

For the `datePart` parameter, you can specify `HOUR`, `MINUTE`, `SECOND`, `DAY_OF_WEEK`, `DAY_OF_MONTH`, `MONTH`, or `YEAR`.

The following sample Gosu codes uses the `DatePart` method to extract the day of the month from the due date on activities. The query builder codes uses the returned numeric value in a comparison predicate to select activities that are due on the 15th of any month.

```
uses gw.api.database.Query
uses gw.api.database.DBFunction

var query = Query.make(Activity)
// Query for activities with due dates that fall on the 15th of any month.
query.compare(DBFunction.DatePart(DAY_OF_MONTH, query.getColumnRef("endDate")), Equals, 15)

// Order the result by assignment date and iterate the items fetched.
for (activity in (query.select().orderBy(\ row -> row.AssignmentDate))) {
    print("Assigned on " + activity.AssignmentDate + ": " + activity.DisplayName)
}
```

When you specify `DAY_OF_WEEK` as the part to extract, the first day of the week is Monday.

Comparing the Date Part of a Date and Time Field

Use the static `DateFromTimestamp` method of the `DBFunction` class to extract the date portion of a `DateTime` field to use in a comparison predicate. The `DateFromTimestamp` method takes a single parameter, a column reference to a `DateTime` field. It returns a `java.util.Date` with only the date portion specified.

```
DateFromTimestamp(timestamp : ColumnRef) : DBfunction
```

The following sample Gosu code uses the `DateFromTimestamp` method to extract the date from the create timestamp on activities. The query builder code uses the returned date in a comparison predicate to select activities that were created some time during the current day.

```
uses gw.api.database.Query
uses gw.api.database.DBFunction

// Make a query of Address instances.
var query = Query.make(Address)

// Query for addresses created today.
query.compare(DBFunction.DateFromTimestamp(query.getColumnRef("CreateTime")), Equals, DateTime.Today)

// Order the result by creation date and iterate the items fetched.
for (Address in query.select().orderBy(\ row -> row.CreateTime)) {
    print(address.DisplayName + ": " + address.CreateTime)
}
```

Using Comparison Predicates with Null Values

In a relational database, you can define columns that allow null values or that require every row to have a value. In a ClaimCenter application database, you can define entity properties that allow null values or 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 database
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 48

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.

Using Set Inclusion and Exclusion Predicates

In SQL, you can embed an SQL query in-line within another query. This technique is called a *subselect* or a *subquery*. The SQL query language does not provide a keyword to create a subselect. You create one by the way in which you use the SQL query syntax. The following SQL statement creates a subselect to provide a set of values for an IN predicate.

```
SELECT column_name1 FROM table_name
WHERE column_name1 IN (SELECT column_name2 FROM table_name2
WHERE column_name2 LIKE '%some_value%');
```

In comparison with the preceding SQL example, the following Gosu sample code constructs and executes a functionally equivalent query.

```
uses gw.api.database.Query

var query = Query.make(SomeEntity)
query.subselect("SomePropertyName", CompareIn, SomeOtherEntity, "SomeOtherProperty")

var result = query.select()
```

The `subselect` method is overloaded with many method signatures for different purposes. The preceding Gosu sample code uses the simplest of the `subselect` method signatures.

The query builder APIs generate an SQL `IN` clause for the `CompareIn` method. The query builder APIs generate an SQL `NOT EXISTS` clause for the `CompareNotIn` method.

Comparing Column Values with Each Other

Some predicate methods have signatures that let you compare a column value for an entity instance with another column value for the same instance. Use the `getColumnRef` method on query, table, and restriction objects to obtain a column reference that you can pass to comparison methods that take them as arguments.

For example, you want to query the database for contacts where the primary and secondary email addresses differ. The following SQL statement compares the two email address columns on a contact row with each other.

```
SELECT * FROM Contacts  
WHERE city = EmailAddress1 NOT EQUAL EmailAddress2;
```

In comparison with the preceding SQL example, the following Gosu sample code uses the `getColumnRef` method to compare the two email address properties on a contact instance with each other.

```
uses gw.api.database.Query  
  
var query = Query.make(Contact) // Query for contacts where email 1 and email 2 do not match.  
query.compare("EmailAddress1", NotEquals, query.getColumnRef("EmailAddress2"))  
  
var result = query.select()  
  
for (contact in result) { // Execute the query and iterate the results  
    print("public ID " + contact.PublicID + " with name " + contact.DisplayName  
        + " and emails " + contact.EmailAddress1 + "," + contact.EmailAddress2)  
}
```

Predicate method signatures that let you compare column values with another take column names directly as the first parameter. Use the `getColumnRef` method to acquire a value for the third parameter only.

Note: The `ColumnRef` objects that the `getColumnRef` method returns does not implement the `IQueryBuilder` interface. So, you cannot chain the `getColumnRef` method with other query builder methods. For more information, see “Chaining Query Builder Methods” on page 174.

Comparing Column Values with Literal Values

When you pass Gosu literals as comparison values to predicate methods, the generated SQL sent to the relational database treats them as SQL query parameters. Use the `DBFunction.Constant` static method to specify literal values to treat as SQL literals in the SQL query that ClaimCenter submits to the relational database.

If your query builder code uses the same Gosu literal in all invocations, replace the Gosu literal with a call to the `DBFunction.Constant` method. Use of the method to coerce a Gosu literal to an SQL literal can improve query performance, provided that:

- You compare the literal value to a column that is in an index, and one of the following:
 - Your query builder code always passes the same literal value.
 - Your literal value is either sparse or predominant in the set of values in the database column.

IMPORTANT Use the `DBFunction.Constant` method with caution. If you are unsure, consult your database administrator for guidance.

Differences between Gosu Literals and Database Constants

Gosu Literal in a Query Example 1

The following sample Gosu code uses the Gosu literal “Los Angeles” to select addresses from the application database.

```
uses gw.api.database.Query  
  
// Query for addresses by using a Gosu literal to select addresses in Los Angeles.  
var query = Query.make(Address)  
query.compare("City", Equals, "Los Angeles")  
  
var result = query.select()
```

```

for (Address in result) {
    print("public ID " + address.PublicID + " with city " + address.City)
}

```

The code specifies a literal value for the predicate comparison, but the SQL query that the code generates uses the value "Los Angeles" as a prepared parameter.

SQL Literal in a Query Example 2

In contrast, the following sample code uses the `DBFunction.Constant` method to force the generated SQL query to use the Gosu string literal "Los Angeles" as an SQL literal.

```

uses gw.api.database.Query
uses gw.api.database.DBFunction

// Query for addresses by using an SQL literal to select addresses in Los Angeles.
var query = Query.make(Address) // Query for addresses.
query.compare("City", Equals, DBFunction.Constant("Los Angeles"))

var result = query.select()

for (Address in result) {
    print("public ID " + address.PublicID + " with city " + address.City)
}

```

Constant Method Signatures and the Predicate Methods that Support Them

The `DBFunction.Constant` method has these signatures:

```

Constant(value String) : DBFunction
Constant(value Number) : DBFunction
Constant(dataType IDataType, value Object) : DBFunction

```

The predicate methods that support a `DBFunction` as the comparison value are as follows:

- `compare`
- `compareIgnoreCase`
- `between`
- `startsWith`
- `contains`
- `subselect`

Comparing Typekey Column Values with Typekey Literals

Often you want to select data based on the values of *typekey* fields. A typekey field takes its values from a specific *typelist*, which contains a set of codes and related display values used in the drop-down lists of the ClaimCenter application.

Gosu provides *typekey literals* that let you specify typelist codes in your programming code. Gosu creates typekey literals at compile time by prefixing typelist codes with `TC_` and converting code values to upper case. For example, the following Gosu sample code specifies the typekey code for "open" from the `ActivityStatus` typelist.

```
typekey.ActivityStatus.TC_OPEN
```

The `Address` entity type has a typekey field named `State` that takes its values from the `State` typelist. The following sample code uses the typekey literal for California to select and print all addresses in California.

```

uses gw.api.database.Query

// Query the database for addresses.
var query = Query.make(Address)

// Restrict the query to addresses in the state of California.
query.compare("State", Equals, typekey.State.TC_CA)

var result = query.select()

// Iterate and print the result.
for (Address in result) {

```

```
    print(address.AddressLine1 + ", " + address.City + ", " + address.State)
}
```

Use code completion in Studio to build complete object path expressions for typelist literals. To begin, type `typekey`, and then work your way down through the typelist name to the typekey code that you want.

See also

- “Typekey Literals” on page 56
- “Working with Typelists” on page 271 in the *Configuration Guide*

Combining Predicates with AND and OR Logic

Often you need more than one predicate in your query to select the items that you want in your result. For example, to select someone named “John Smith” generally requires you to specify two predicate expressions:

```
last_name = "Smith"  
first_name = "John"
```

To select the person you seek, you want both expressions to be true. That is, you want a person’s last name to be “Smith” *and* you want that same person’s first name to be “John.” You do not want either expression to be true. That is, you do not want to select people whose last names are “Smith” *or* first names are “John.”

In contrast with the preceding example, there are times when you want to combine predicate expressions, any of which must be true to select the items that you want. For example, to select addresses from Chicago or Los Angeles also generally requires two predicates:

```
city = "Chicago"  
city = "Los Angeles"
```

To select the addresses that you seek, you want either expression to be true. That is, you want an address to have “Chicago” as the city *or* you want it to have “Los Angeles.”

Using AND to Combine Predicates that All Must Be True

In SQL, you combine predicates with the AND keyword if *all* must be true to include an item in the result.

```
WHERE last_name = "Smith" AND first_name = "John"
```

With the query builder APIs, you combine predicates by calling predicate methods on the query object one after the other if *all* must be true.

```
query.compare("LastName", Equals, "Smith")  
query.compare("FirstName", Equals, "John")
```

Using OR to Combine Predicates that Only One of which Must Be True

In SQL, you combine predicates with the OR keyword if *only one* must be true to include an item in the result.

```
WHERE city = "Chicago" OR city = "Los Angeles"
```

With the query builder APIs, you combine predicates by calling the OR method on the query object if *only one* must be true.

```
query.or( \ orCriteria -> {  
    orCriteria.compare("City", Equals, "Chicago")  
    orCriteria.compare("City", Equals, "Los Angeles")  
}
```

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 231.

The default behavior of a new query 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. However, the way you apply and and or methods to a query can change the default linking mode of a new series of predicates.

The and and or 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.

The or method has the following effect on the predicate linking mode of query:

- To previously specified restrictions, add one parenthetical phrase.
- Link this new parenthetical phrase to previous restrictions in the linking mode already in effect.
- The block passed to the or method includes a series of predicates.
- For 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 AND or OR grouping, each created with another usage of the and or or methods.

The and method has the following effect on the predicate linking mode of query:

- To the previous specified restrictions, add one parenthetical phrase.
- Link this new parenthetical phrase to previous restrictions in the linking mode already in effect.
- The block passed to the and method 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 AND or OR 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.PredateOrBooleanGrouping(...)
    OR_GROUPING_VAR.PredateOrBooleanGrouping(...)
    OR_GROUPING_VAR.PredateOrBooleanGrouping(...)
    [...]
}
```

The syntax of the and method is

```
query.and( \ AND_GROUPING_VAR -> {
    AND_GROUPING_VAR.PredateOrBooleanGrouping(...)
    AND_GROUPING_VAR.PredateOrBooleanGrouping(...)
    AND_GROUPING_VAR.PredateOrBooleanGrouping(...)
    [...]
}
```

Each use of *PredateOrBooleanGrouping* 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 words “and” and “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, where 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

If your query consists of a large number of predicates linked by OR clauses, there is an alternative approach that might provide better performance in production, 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 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 your query with both approaches, and then test production performance with a large number of records.

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 method chaining of query builder APIs to make it fit 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 using this pattern in the example, and avoid chaining the predicates together.

Predicate Methods Reference

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	<ul style="list-style-type: none"> • Column name (String) • Operation type (Relop) • Value (Object) 	<p>Compares a column to a value. For the operation type, pass one of the following values to represent the operation type:</p> <ul style="list-style-type: none"> • 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. <p>Pass these values without quote symbols around them. These names are values in the Relop enumeration.</p> <p>For the value object, you can use numeric types, String types, ClaimCenter entities, keys, or typekeys. For String values, the comparison is case-sensitive.</p> <p>Example of a simple equals comparison:</p> <pre>query.compare("Priority", Equals, 5)</pre> <p>Example of a simple less than or equal to comparison:</p> <pre>query.compare("Priority", LessThanOrEquals, 5)</pre> <p>To compare the value to the value in another column, generate a column reference and pass that instead. See "Comparing Column Values with Each Other" on page 145 for extended discussion on this subject.</p> <p>You can use algebraic functions that evaluate to an expression that can be evaluated at run time to be the appropriate type. For example:</p> <pre>var prefix = "abc:" // string variable... var recordNumber = "1234" query.compare("PublicID", Equals, prefix + recordNumber)</pre> <p>Or combine a column reference and algebraic functions:</p> <pre>query.compare("Priority", Equals, DBFunction.Expr({query.getColumnRef("OldPriority") }, "+", "10"))</pre>
compareIgnoreCase	<ul style="list-style-type: none"> • Column name (String) • Operation type (Relop) • Value (Object) 	<p>Compares a character column to a character value while ignoring uppercase and lowercase variations. For example, if the following comparison succeeds:</p> <pre>query.compare("Name", Equals, "Acme")</pre> <p>The following comparison also succeeds:</p> <pre>query.compareIgnoreCase("Name", Equals, "ACME")</pre>

Predicate method	Arguments (Type)	Description
startsWith	<ul style="list-style-type: none"> Column name (String) Substring value (String) Ignore case (Boolean) 	<p>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:</p> <pre>query.startsWith("FirstName", "jo", true /* ignore case */)</pre> <p>Note: If you choose case insensitivity, Gosu generates an SQL LOWER function to implement the comparison predicate. However, if the data model definition of the column specifies the supportsLinguisticSearch attribute set to true, Gosu uses the denormalized version of the column, instead.</p>
contains	<ul style="list-style-type: none"> Column name (String) Contains value (String) Ignore case (Boolean) 	<p>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:</p> <pre>query.contains("FirstName", "jo", true /* ignore case */)</pre>
compare	<ul style="list-style-type: none"> Range (a Range subclass) 	<p>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. <i>Refer to the following three rows which list the range subclasses.</i></p>
compare	<ul style="list-style-type: none"> Number range (NumberRange) 	<p>If the column type is a number (its type extends from Number), use the range subclass NumberRange. The following code defines a range of numbers:</p> <pre>query.compare("Priority", new NumberRange(5,10))</pre> <p>To specify an unbounded range on the lower or upper end, pass null to the constructor as the first or second range argument but not both. You can pass null as a range argument regardless of null restrictions on the column.</p>
compare	<ul style="list-style-type: none"> Date range (DBDateRange) 	<p>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 the date range January 1, 1990, to January 1, 2010.</p> <pre>query.compare("DateOfBirth", new DBDateRange(new DateTime(90,0,1), new DateTime(110,0,1), true))</pre> <p>To specify an unbounded range on the lower or upper end, pass null to the constructor as the first or second range argument but not both. You can pass null as a range argument regardless of null restrictions on the column.</p>
compare	<ul style="list-style-type: none"> Value range. Use with text values or anything other than date or number. (ValueRange) 	<p>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, the following code defines a range of String values to match:</p> <pre>query.compare("PublicID", new ValueRange<String>("abc:1","abc:99"))</pre> <p>To specify an unbounded range on the lower or upper end, pass null to the constructor as the first or second range argument but not both. You can pass null as a range argument regardless of null restrictions on the column.</p>

Predicate method	Arguments (Type)	Description
between	<ul style="list-style-type: none"> Column name (String) Start value (Object) End value (Object) 	<p>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.</p> <pre>query.between("PublicID", "abc:01", "abc:99")</pre> <p>To specify an unbounded range on the lower or upper end, pass null as the first or second range argument but not both. You can pass null as a parameter regardless of null restrictions on the column.</p>
compareIn	<ul style="list-style-type: none"> Column name (String) List of values that could match the database row for that column (Object[]) 	<p>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:</p> <pre>query.compareIn("PublicID", {"default_data:1", "default_data:3"})</pre>
compareNotIn	<ul style="list-style-type: none"> Column name (String) List of values that could match the database row for that column (Object[]) 	<p>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:</p> <pre>query.compareNotIn("PublicID", {"default_data:1", "default_data:3"})</pre>
subselect	See "Using Set Inclusion and Exclusion Predicates" on page 144.	Perform a join with another table and select a subset of the data by combining tables. For more information, see "Using Set Inclusion and Exclusion Predicates" on page 144.
or	Gosu block that contains a list of predicate methods applied to columns in the query.	Checks whether a value satisfies one or more predicate methods, such as compare, contains, and between. Only one of the predicate methods must evaluate to true for the item that contains the value to be included in the result. For more information, see "Combining Predicates with AND and OR Logic" on page 147
and	Gosu block that contains a list of predicate methods applied to columns in the query.	Checks whether a value satisfies a set of predicate methods, such as compare, contains, and between. All of the predicate methods must evaluate to true for the item that contains the value to be included in the result. For more information, see "Combining Predicates with AND and OR Logic" on page 147

Working with Row Queries

You cannot achieve certain kinds of SQL results with entity queries. For example, entity queries cannot achieve the results of SQL aggregate queries. Neither can entity queries achieve results that involve SQL outer joins. In both cases, use row queries, instead. In addition, sometimes you need only a few columns in the result, not the complete entity object graph. In such cases, use row queries to select and return only the columns of data that you need.

Setting Up Row Queries

You set up a row query by calling the `select` method on query objects that accepts a *block* of column selections. In Gosu, a block is an in-line function that you define within another function. The block that you pass to the `select` method 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:

```
queryObject.select(\ resultRow -> (
    {"key1" -> resultRow.objectPathExpression1[},   // The block argument is a selected row.
     "key2" -> resultRow.objectPathExpression2[}, ...
) // end of hash map declaration
) // and block body
) // end of select method
```

Gosu inspects the block passed to the `select` method at compile time to determine which database columns provide results to 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.

Gosu also inspects the block passed to the `select` block at run time to determine which properties to load. As you iterate across the result set, after a row loads from the database, Gosu runs your block. For each item in the result, Gosu executes your block and replaces the values of column names with values for the item from the database.

The use of the `select` method that accepts a Gosu block of column selections is available only on query objects, not tables or restrictions. You cannot use virtual properties, enhancement methods, or other entity methods within the block that you pass to the `select` method. Refer to the *Data Dictionary* to determine which properties are backed by database columns.

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 refer to properties backed by actual database columns.
- 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 three database columns into a string array, you could use code similar to the following:

```
uses gw.api.database.Query

// Define constants for each selected column in the row array.
var FirstName = 0
var LastName = 1
Var State = 2

var query = Query.make(Person)

// Select columns from the Person table to return in the result.
var resultRows = query.select(\ row -> {return {
    row.FirstName, // element 0 in the returned row array
    row.LastName, // element 1 in the returned row array
    row.PrimaryAddress.State as String // element 2 in the returned row array
}

// Return the selected columns for each select row as String array.
} as String[] })

for (row in resultRows) {
    // Print the array elements for the selected columns the current result row.
    print (row[FirstName] + " " + row[LastName] + ", " + row[State])
}
```

In the previous example, note the references to `row.FirstName`, `row.LastName`, and `row.PrimaryAddress.State`. 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
}
```

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`

Gosu permits the line with a computation, `row.A + 10`, as shown in the example below. Gosu does not use an SQL feature called *computed columns* to optimize rows with computations in the relational database. Instead, Gosu loads the column values 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.

IMPORTANT If your block applies complex calculations to a row, test and profile query performance with large data loads to determine the performance impact of your Gosu 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  
        }  
    }  
})
```

IMPORTANT Choose the type you want for the result carefully. Performance characteristics vary depending on the type that you choose.

See also

- “Gosu Blocks” on page 231
- “Generics and Blocks” on page 244
- “How Generics Help Define Collection APIs” on page 246

Database Aggregate Functions Within Select Blocks

The following Gosu sample code uses the `Sum` database aggregate function.

```
uses gw.api.database.Query  
uses gw.api.database.DBFunction  
  
var rows = Query.make(TestA).select(\row-> new HashMap<String, Integer>() {  
    "A" -> DBFunction.Sum(row.A), // Compute the sum of values in column A.  
    "A2" -> row.A2 // Group the rows by differing values in column A2 before summing.  
})
```

The query returns the sum of all values in column A, grouped by differing values in column A2. The query builder APIs generate the implied SQL `GROUP BY` clause on column A2.

Note: The query builder APIs insert a `GROUP BY` clause in the generated SQL whenever you use any of the database aggregate functions in a query builder expression.

Within your block, you can use the following database functions: `Sum`, `Max`, `Min`, `Avg`, and `Count`.

Applying Functions to Selected Fields

Database queries sometimes require native database functions found in SQL. For example, a query might need the power of the SQL minimum and maximum functions that examine rows already in the database.

The query builder APIs includes built-in database functions implemented as methods on the `DBFunction` class. Each time you use these database functions, the query builder APIs generate exactly one corresponding native database function in the SQL that it generates. For example, if you use the `Min` query builder APIs database function to calculate the minimum value in a column, Gosu generates the SQL `MIN` function.

The query builder APIs database functions all take a single argument, which is a column reference. Use the `getColumnRef` method on a query, and pass it the column name as a `String` value.

```
var restriction = query.getColumnRef("IntegerExt")
```

You can only use the database functions in the contexts permitted by the SQL standard. For example, all functions listed in the table below, except for `Constant`, represent data aggregated from multiple rows. In database terms, these functions aggregate data from other rows. These functions are known as *aggregate functions*. You cannot use an aggregate 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 usage limitations of database aggregate functions.

Note: 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 aggregated data from columns in another table.

For example, the following query is valid:

```
query.having().compare(DBFunction.Constant(44), GreaterThanOrEquals,
                      DBFunction.Avg(query.getColumnRef("IntegerExt")))
```

The following query is invalid, because the function compares the column named `A` against an aggregate version of the column from the same table:

```
query.compare("A", Equals, DBFunction.Min(query.getColumnRef("A")))
```

This is also invalid, even if you use different columns on the same table:

```
query.compare("A", Equals, DBFunction.Min(query.getColumnRef("B")))
```

You can work around this SQL limitation buy using one of the following approaches:

- Performing a join on the table. For information about using database functions with joins, see “Joining Related Entities to Queries” on page 130.
- 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 query = Query.make(SampleBean)
var joinTable = query.join(SampleChildBean, "parent")
query.having().between("TextField1", DBFunction.Min(joinTable.getColumnRef("TextField1")),
                      DBFunction.Max(joinTable.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
SQL aggregate functions		
Avg	Returns the average of all values in a column	query.compare(DBFunction.Constant(44), GreaterThan, DBFunction.Avg(query.getColumnRef("B")))
Count	Returns the number of rows in a column	query.compare(DBFunction.Constant(44), GreaterThan, DBFunction.Count(query.getColumnRef("B")))

Function	Description	Example
Min	Returns the minimum value of all values in a column	query.compare(DBFunction.Constant(44), GreaterThan, DBFunction.Min(query.getColumnRef("B"))))
Max	Returns the maximum value of all values in a column	query.compare(DBFunction.Constant(44), LessThan, DBFunction.Max(query.getColumnRef("B"))))
Sum	Returns the sum of all values in a column	query.compare(DBFunction.Constant(44), GreaterThan, DBFunction.Sum(query.getColumnRef("B"))))
Query Builder API functions		
DateDiff	Returns the interval between two date and time columns	query.compare(DBFunction.DateDiff(DAYS, getColumnRef("AssignmentDate"), getColumnRef("EndDate")), LessThan, 15) See "Comparing the Interval Between Two Date and Time Fields" on page 142
DatePart	Returns parts of a date and time column value, such as day or month	query.compare(DBFunction.DatePart(DAY_OF_MONTH, query.getColumnRef("AssignmentDate")), NotEquals, 15) See "Comparing Parts of a Date and Time Field" on page 142.
Constant	Coerces a Gosu literal to an SQL literal in the generated SQL query that ClaimCenter submits to the database	query.compare("City", Equals, DBFunction.Constant("Los Angeles")) See "Comparing Column Values with Literal Values" on page 145.
Expr	Returns a function defined by a list of column reference and character sequences	See "Example of expr Database Function" on page 157.

Example of expr Database Function

The Expr database function returns a function defined by a list of column references and character sequences. The argument to this function is a list that contains only objects of type:

- java.lang.CharSequence – For example, pass a String that contains SQL operators or other functions.
- gw.api.database.ColumnRef – The type that the query.getColumnRef(columnName) method returns.

The query builder APIs concatenate the objects in the list in order specified to form an SQL expression.

For example, the following Gosu sample code 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 IntegerExt 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
```

Limitations of Row Queries

Row queries have the following limitations:

- **Result values are not part of object domain graphs** – You cannot use object path notation with values in results from row queries. The values in results from row queries are not part of object domain graphs.
- **Access only to columns and foreign keys** – You can access only data from columns and foreign keys with row queries. To access data from arrays, virtual properties, one-to-ones, or edge foreign keys, you must use entity queries.
- **Result values are not in the application cache** – Values in the result are in local memory only. They are not loaded into the application cache.
- **Results cannot be updated** – You cannot update the database with changes that you make to data returned by a row query.
- **Cannot be used with list views and detail panels** – You cannot use the results of row queries as the data sources for list views and detail panels in page configuration files.

See also

- “Features Comparison of Entity and Row Queries” on page 160
- “Performance Differences Between Entity and Row Queries” on page 171

Working with Results

The reason to build queries is to use the information they return to your Gosu program. Frequently you use items returned in result objects to display information in the user interface. For example, you might query the database for a list of doctors and display their names in a list.

This topic contains:

- “What Result Objects Contain” on page 158
- “Filtering Results with Standard Query Filters” on page 160
- “Ordering Results” on page 165
- “Useful Properties and Methods on Result Objects” on page 166
- “Determining if a Result Will Return Too Many Items” on page 168
- “Updating Entity Instances in Query Results” on page 170

What Result Objects Contain

The query builder APIs support two types of queries:

- **Entity queries** – Result objects contain a list of references to instances of the primary entity type for the query. You set up an entity query with the `select` method that takes no arguments. For example:

```
query.select()
```

- **Row queries** – Result objects contain a set of row-like structures with values fetched from or computed by the relational database. You set up a row query with the `select` method that takes a Gosu block of column selections. For example:

```
query.select(columnSelectionBlock)
```

Content of Result Sets from Entity Queries

Result objects from entity queries contain a list of references to instances of the primary entity type for the query. For example, you write the following Gosu sample code to begin a new query.

```
var query = Query.make(Company)
```

The preceding sample code sets the primary entity type for the query to `Company`. Regardless of related entity types that you join to the primary entity type, the result contains only instances of the primary entity type, `Company`.

To set up the preceding query as an entity query, call the `select` method without parameters, as the following Gosu sample code shows.

```
var entityResult = query.select() // The select method with no arguments sets up an entity query.
```

The members of results from entity queries are instances of the type from which you make queries. In this example, the members of `entityResult` are instances of `Company`. After you retrieve members from the results of entity queries, use object path notation to access objects, methods, and properties from anywhere in the object graph of the retrieved member.

The following Gosu sample code prints the name and the city of the primary address for each `Company` in the result. The `Name` property is on a `Company` instance, while the `City` property is on an `Address` instance.

```
for (company in entityResult) { // Members of a result from entity query are entity instances.
    print(company.Name + ", " + company.PrimaryAddress.City)
}
```

Contents of Result Sets from Row Queries

Result objects from row queries contain a set of row-like structures that correspond to the column selection block that you pass as an argument to the `select` method. Each structure in the set represents a row in the database result set. The members of each structure contain the values for that row in the database result set.

For example, you write the following Gosu sample code to begin a new query.

```
var query = Query.make(Company)
```

The preceding sample code sets the primary entity type for the query to `Company`. You can join other entity types to the query and specify restrictions, just like you can with entity queries. Unlike entity queries however, the results of row queries do not contain entity instances.

The results of row queries contain values selected and computed by the relational database, based on the column selection block you pass to the `select` method. To set up the preceding query as a row query, call the `select` method and pass a Gosu block that specifies the columns you want to select for the result.

```
var rowResult = query.select(\ row -> \{ // Select methods with a block argument set up row queries.
    "Name" -> row.Name,
    "City" -> row.PrimaryAddress.City
})
```

The members of results from row queries are structures that correspond to the column selection block you specify.

After you retrieve a member from the result of a row query, you can only use the values the member contains. You cannot use object path notation with a retrieved member to access objects, methods, or properties from the domain graph of the primary entity type of the query.

The following Gosu sample code prints the company name and the city of the primary address for each Company in the result. The Name property and the City property are properties of the result member.

```
for (company in rowResult) { // Members of a result from entity query are entity instances.
    print (company.get("Name") + ", " + company.get("City"))
}
```

You can only access properties from the members of a row result if you specified them in the column selection block.

Features Comparison of Entity and Row Queries

The following table compares features of entity queries and row queries.

Feature	Entity queries	Row queries
Result contents	Results from entity queries contain references to entity instances, so you can use object path notation to access objects, properties, and methods in their object graphs.	Results from row queries contain values with no access to the object graphs of the entity instances that matched the query criteria.
Entity field types	Entity queries can access data from columns, foreign keys, arrays, virtual properties, one-to-ones, and edge foreign keys.	Row queries can access data only from columns and foreign keys.
Application cache	Entity instances in the result are loaded into the application cache.	Values in the result are in local memory only. They are not loaded into the application cache.
Writable results	Entity instances returned in results can be changed in the database by moving them from the result to a writable bundle.	Results cannot be used directly to write changes to the database.
Page configuration	Results can be data sources for list views and detail panels in page configuration.	Results cannot be data sources for list views or detail panels in page configuration directly.

See also

[“Performance Differences Between Entity and Row Queries” on page 171](#)

Filtering Results with Standard Query Filters

Sometimes you want to use a query result object and filter the items in different ways when you iterate them. For example, you have a query with complicated joins, predicates, and subselects. Several routines want to use the results of this complex query, but each routine processes different subsets of the results.

In such cases, you can separate the query from the filtering of its results. You write query builder code in your main routine to construct and refine the query and then produce a result object with the `select` method. If you build the query once in your main routine, you can debug and maintain the query logic of your overall programming unit more easily.

After you obtain a result object in your main routine, pass the result object to each subroutine. The subroutines apply their own, more restrictive predicates by using standard query filters. Also, the subroutines can apply their own ordering and grouping requirements. ClaimCenter combines the query predicates from the main routine and the standard query filter predicates from the subroutine to form the SQL query that it sends to the relational database.

In addition to using standard query filters in Gosu code, you can use standard query filters in the page configuration of your ClaimCenter application. List views support a special type of toolbar widget, a toolbar filter. A

toolbar filter lets users choose from a drop-down menu of filters to apply to the set of data that a list view contains. Toolbar filters accept different kinds of filters, including standard query filters. For more information, see “Using Standard Query Filters in Toolbar Filters” on page 163.

Creating a Standard Query Filter

A *standard query filter* represents a named query predicate that you can add to a query builder result object. You create a standard query filter by instantiating a new `StandardQueryFilter` object. The `gw.api.filters` package contains the `StandardQueryFilter` class. Its constructor takes two arguments:

- **Name** – A String to use as an identifier for the filter.
- **Predicate** – A Gosu block with a query builder predicate method. You must apply the predicate method to a field in the query result that you want to filter. You can use the same predicate methods in standard query filter predicates that you use on queries themselves.

The following sample Gosu code creates a standard query filter that can apply to query results that include `Address` instances. The standard query filter predicate uses a simple `compare` predicate method on the `City` field.

```
var myQueryFilter = new StandardQueryFilter("myQueryFilter",
    \ query -> {query.compare("City", Equals, "Bloomington")})
```

Note: The package `gw.api.filters` contains pre-defined standard query filters that you can apply as needed.

Adding a Standard Query Filter to a Query Result

Use the `addFilter` method on a query builder result object to restrict the items you obtain when you iterate the result. The method takes as its single argument a `StandardQueryFilter` object. You can add as many filters as you want to a query result.

The following sample Gosu code adds a standard query filter to restrict a result object to addresses in the city of Chicago.

```
uses gw.api.database.Query
uses gw.api.filters.StandardQueryFilter

// Create a query of addresses.
var result = Query.make(Address).select()

// Create a standard query filter for addresses in the city of Chicago.
var queryFilterChicago = new StandardQueryFilter("Chicago Addresses",
    \ query -> {query.compare("City", Equals, "Chicago")})

// Add the Chicago addresses filter to the result.
result.addFilter(queryFilterChicago)

// Iterate the addresses in Chicago.
for (Address in result) {
    print (address.City + ", " + address.State + " " + address.PostalCode)
}
```

Use the `addFilter` method when you want a single result to have one or more query filters in effect at the same time.

Using AND and OR Logic with Standard Query Filter Predicates

You can use ‘AND’ and ‘OR’ logic with standard query filter predicates in the same way that you use AND’ and ‘OR’ logic with predicates on a query builder query.

- **For AND logic** – Add multiple standard query filters, each with a single comparison predicate, to a result.
- **For OR logic** – Add a single standard query filter with an `or` method in its filter predicate to a result.

For more information, see “Combining Predicates with AND and OR Logic” on page 147.

Using AND Logic with Standard Query Filter Predicates

With standard query filters, you combine filter predicates that *all* must be true by adding standard query filters to a result one after the other. The following sample Gosu code adds two standard query filters to a result object. The first filter restricts the iteration to addresses in the city of Chicago. The second filter further restricts the iteration to addresses that were added to the database during 2012 or later.

```
uses gw.api.database.Query
uses gw.api.filters.StandardQueryFilter

// Create a query of addresses.
var result = Query.make(Address).select()

// Create a standard query filter for addresses in the city of Chicago.
var queryFilterChicago = new StandardQueryFilter("Chicago Addresses",
    \ query -> {query.compare("City", Equals, "Chicago")})

// Create a standard query filter for addresses added during 2012 or later.
var queryFilter2012Address = new StandardQueryFilter("2012 Addresses",
    \ query -> {query.compare("CreateTime", GreaterThanOrEquals,
        "2012-01-01" as java.util.Date)})

// Add the Chicago and 2012 address filters to the result.
result.addFilter(queryFilterChicago)
result.addFilter(queryFilter2012Address)

// Iterate the addresses in Chicago added during 2012 or later.
for (Address in result) {
    print(address.City + ", " + address.State + " " + address.PostalCode + " " + address.CreateTime)
}
```

If you add more than one standard query filter to a result, make sure that each filter predicate applies to a different field in the result. If you add two or more standard query filters with predicates on the same field, Boolean logic ensures that no item in the result satisfies them all.

Using OR Logic with Standard Query Filter Predicates

With standard query filters, you combine predicates only one of which must be true by using the `or` method in the filter predicate of a single standard query filter. The following sample Gosu code adds two predicates to a standard query filter to restrict the iteration of addresses to the cities of Bloomington or Chicago.

```
uses gw.api.database.Query
uses gw.api.filters.StandardQueryFilter

// Create a query of addresses.
var result = Query.make(Address).select()

// Create a standard query filter for addresses in the cities of Bloomington or Chicago.
var queryFilterBloomingtonOrChicago = new StandardQueryFilter("Bloomington and Chicago Addresses",
    \ query -> {query.or(\ orCriteria -> {
        orCriteria.compare("City", Equals, "Bloomington")
        orCriteria.compare("City", Equals, "Chicago")
    })
})

// Add the Bloomington or Chicago filter to the result.
result.addFilter(queryFilterBloomingtonOrChicago)

// Iterate the addresses in Bloomington or Chicago, in order by city.
for (Address in result.orderBy(\ row -> row.City)) {
    print(address.City + ", " + address.State + " " + address.PostalCode)
}
```

Using Standard Query Filters in Gosu Code

You often use standard query filters in Gosu code to separate code that constructs a general purpose query from code in subroutines that process different subsets of the result. Query builder code that constructs a query can be complex whenever joins, subselects, and compound predicate expressions are involved. Locating this part of your query builder code in one location makes debugging and maintenance easier.

The following sample Gosu code builds a query for addresses in the state of Illinois. Then, the code passes the query result to three subroutines for processing. Each subroutine creates and adds a standard query filter for a

city in Illinois: Bloomington, Chicago, or Evanston. Then, the subroutines iterate their own subset of the main query result.

```
uses gw.api.database.Query
uses gw.api.database.IQueryBeanResult
uses gw.api.filters.StandardQueryFilter

// Select addresses from the state of Illinois.
var query = Query.make(Address)
query.compare("State", Equals, typekey.State.TC_IL)

// Get a result and apply ordering
var result = query.select()
result.orderBy(\ row -> row.City).thenBy(\ row -> row.PostalCode)

// Pass the ordered result to subroutines to process Illinois addresses by city.
processBloomington(result)
result.clearFilters() // Remove any filters added by the subroutine from the result.

processChicago(result)
result.clearFilters() // Remove any filters added by the subroutine from the result.

processEvanston(result)
result.clearFilters() // Remove any filters added by the subroutine from the result.

// subroutines

// Add Bloomington filter and process results.
function processBloomington(aResult : IQueryBeanResult<Address>) {
    var queryFilterBloomington = new StandardQueryFilter("QueryFilterBloomington",
        \ query -> {query.compare("City", Equals, "Bloomington")})
    aResult.addFilter(queryFilterBloomington)
    for (address in aResult) {
        print (address.City + ", " + address.State + " " + address.PostalCode)
    }
}

// Add Chicago filter and process results.
function processChicago(aResult : IQueryBeanResult<Address>) {
    var queryFilterChicago = new StandardQueryFilter("QueryFilterChicago",
        \ query -> {query.compare("City", Equals, "Chicago")})
    aResult.addFilter(queryFilterChicago)
    for (address in result) {
        print (address.City + ", " + address.State + " " + address.PostalCode)
    }
}

// Add Evanston filter and process results.
function processEvanston(aResult : IQueryBeanResult<Address>) {
    var queryFilterEvanston = new StandardQueryFilter("QueryFilterEvanston",
        \ query -> {query.compare("City", Equals, "Evanston")})
    aResult.addFilter(queryFilterEvanston)
    for (address in result) {
        print (address.City + ", " + address.State + " " + address.PostalCode)
    }
}
```

Using Standard Query Filters in Toolbar Filters

You often use standard query filters with list views in the page configuration of the application user interface. The row iterators of list views support toolbar filter widgets. A toolbar filter lets users select from a drop-down menu of query filters to view subsets of the data that the list view displays. Standard query filters are one type of query filter that you can add.

You specify the standard query filters for a toolbar filter on the **Filter Options** tab. On the tab, you can add two kinds of filter options:

- **ToolbarFilterOption** – An expression that resolves to a single object that implements the `BeanBasedQueryFilter` interface, such as a standard query filter.
- **ToolbarFilterOptionsGroup** – An expression that resolves to an array of objects that implement the `BeanBasedQueryFilter` interface, such as standard query filters.

You can specify a standard query filter or an array of standard query filters by using an inline constructor in the `filter` property of a filter option. Alternatively, you can specify a Java or Gosu class that returns a standard query filter or an array of them.

Example of a Single Toolbar Filter Option

The following example `filter` property specifies a standard query filter by using an inline constructor. The filter applies to activity patterns that the `ActivityPatternsLV` list view displays.

```
ToolbarFilterOption
filter*           |new gw.api.filters.StandardQueryFilter("Mandatory", \ filter ->
{filter.compare("Mandatory", Equals, true)})
```

Note: The previous code block for the single-line filter option field in the PCF editor contains line breaks and extra spaces for readability. If you copy and paste this code, remove these line breaks and spaces to make the code valid.

The toolbar filter displays the first parameter, “Mandatory”, as an option in the toolbar filter drop-down menu.

For single filter options, you can override the text of the drop-down menu of with the `label` property. For localization purposes, you must specify the filter name or the label property as a display key, not as a `String` literal.

Example of a Group Toolbar Filter Option

The following example `filter` property specifies an array of two standard query filters by using inline constructors.

```
ToolbarFilterOption
filter*           |new gw.api.filters.StandardQueryFilter[] {new
gw.api.filters.StandardQueryFilter("Mandatory", \ query ->
{query.compare("Mandatory", Equals, true)}),
new gw.api.filters.StandardQueryFilter("Optional", \ query ->
{query.compare("Optional", Equals, false)})}
```

Note: The previous code block for the single-line filter option field in the PCF editor contains line breaks and extra spaces for readability. If you copy and paste this code, remove these line breaks and spaces to make the code valid.

The toolbar filter displays the first parameters of the filters, “Mandatory” and “Optional”, as options in the toolbar filter drop-down menu. The drop-down menu displays them together and in the order that you specify in the array constructor.

Group filter options do not have a `label` property, so the text of the menu options comes only from the filter names. For localization purposes, you must specify the filter names as display keys, not as `String` literals.

Toolbar Filter Caching

Generally, list views cache the most recent toolbar filter selection made by user in the user’s sessions. If a user leaves a page and then returns to it, a list view retains and applies the toolbar filter option in effect when the user left the page.

You disable filter caching by setting the `cacheKey` property of a toolbar filter to an expression that evaluates to a potentially different value each time a user enters the page. For example, you might specify the following Gosu expression.

```
claim.ClaimNumber
```

If you disable filter caching, the list view reverts to the default filter option for entry to the page. You specify the default filter option by setting the `selectOnEntry` property on the option to `true`. Alternatively, you specify the default filter option by moving the option to the top of the list of options on the **Filter Options** tab.

Toolbar Filter Recalculation

Generally, list views recalculate their filter options only once, when a user enters a page. Filter options remain unchanged during the life span of a page. Sometimes, you may want a list view to recalculate its filter options in response to changes that a user makes on a page.

You force a list view to recalculate its filter options in response to changes by setting the `cacheOptions` property of a toolbar filter to `false`. Set the property to `false` with caution, because the recalculation of filter options after a user makes changes can reduce the speed at which the application renders the updated page.

Ordering Results

By default, SQL Select statements and the query builder APIs return results without regard to the order of the results. However, results in random order might not be useful. So as an option, SQL and the query builder APIs let you specify the order of items in the results they return.

With SQL, the `ORDER BY` clause lets you specify how the database sorts fetched data. The following SQL statement sorts the result set on the postal codes of the addresses.

```
SELECT * FROM addresses
  WHERE city = "Chicago"
    ORDER BY postal_code;
```

In comparison with the preceding SQL example, the following Gosu sample code uses the `orderBy` ordering method to sort addresses in the same way.

```
uses gw.api.database.Query

var queryAddresses = Query.make(Address) // Query for addresses in Chicago.
queryAddresses.compare("City", Equals, "Chicago")
var resultAddresses = queryAddresses.select()

resultAddresses.orderBy(\ row -> row.PostalCode) // Sort the result by postal code.

var iterator = resultAddresses.iterator() // Execute the query and iterate the ordered results.
```

The query builder APIs use the object path expressions that you pass to ordering methods to generate the `ORDER BY` clause for the SQL query. Only when you begin to iterate a result object does Gosu submit the query to the database. The database fetches the items that match the query predicates, sorts the fetched items according to the ordering methods, and returns the result items in that order.

Ordering Methods of the Query Builder APIs

The Query builder APIs support the following ordering methods on result objects.

Method	Description
<code>orderBy</code>	Clears all previous ordering, then orders results by the specified column in ascending order.
<code>orderByDescending</code>	Clears all previous order methods, and orders by the specified column in descending order.
<code>thenBy</code>	Orders by the specified column in ascending order. Does not clear previous ordering.
<code>thenByDescending</code>	Orders by the specified column in descending order. Does not clear previous ordering.

The ordering methods all take a Gosu block as their one argument. The Gosu block has the following syntax.

```
\ row -> row.[ObjectAccessPath.]SimpleProperty
```

The block variable is named `row`, by convention. It represents an instance of the primary entity in the query result. The block code is a Gosu expression that yields an object access path from the primary entity to a simple, non-foreign key, database-backed property.

For example, the following sample Gosu block specifies a simple property, the `PostalCode` on an `Address` instance.

```
\ row -> row.PostalCode
```

At compile time, Gosu checks property names in blocks against column names in the *Data Dictionary*.

Ordering Query Results on Related Instance Properties

The ordering methods of the query builder APIs let you order results on the properties of related entities. To do so, specify an object access path from the primary entity of the query, which the block variable `row` represents. The access path can traverse only database-backed foreign keys, and it must end with a simple, database-backed property. The access path cannot include virtual properties, methods, or calculations. You do not need to join the related entities to your query to reference them in the Gosu blocks that you pass to the ordering methods.

The following sample Gosu code orders notes, based on the date that the activity related to a note was last viewed.

```
uses gw.api.database.Query

var queryNotes = Query.make(Note) // Query for notes.
var resultNotes = queryNotes.select()

result.orderBy(\row-> row.Activity.LastViewedDate) // Sort the notes by related date on activity.
```

Multiple Levels of Ordering Query Results

Often you need to order results on more than one column or property. For example, you query addresses and want them ordered by city. Within a city, you want them ordered by postal code. Within a postal code, you want them ordered by street. In this example, you want three levels of ordering: city, postal code, and street.

The following SQL statement specifies three levels of ordering for addresses.

```
SELECT * FROM addresses
ORDER BY city, postal_code, address_line1;
```

In comparison with the preceding SQL example, the following Gosu sample code constructs and executes a functionally equivalent query.

```
uses gw.api.database.Query

var queryAddresses = Query.make(Address) // Query for addresses.
var resultAddresses = queryAddresses.select()

resultAddresses.orderBy(\ row -> row.City)      // Sort results by city.
resultAddresses.thenBy(\ row -> row.PostalCode) // Within a city, sort results by postal code.
resultAddresses.thenBy(\ row -> row.AddressLine1) // Within a postal code, sort results by street.
```

You can call the ordering methods `thenBy` and `thenByDescending` as many times as you need.

Locale Sensitivity for Ordering Query Results

The query builder APIs order query results by using locale-sensitive comparisons automatically. In contrast, collection enhancement methods for ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Those methods do not sort `String` values in a locale-sensitive way.

Useful Properties and Methods on Result Objects

Query result objects have these useful properties:

- `Empty` – Lets you determine whether a query failed to fetch any matching items.
See “Determining Whether a Query Returned No Results” on page 167
- `Count` – Returns the number of items in the result.
See “Result Counts and Dynamic Queries” on page 167.
- `AtMostOneRow` – Lets you specify that you want the query to return a result only if a single item matches the query predicates.
See “Returning Only Unique Items” on page 168.
- `FirstResult` – Lets you access the first item in a result without iterating the result object.
See “Accessing the First Item in a Result” on page 168.
- `getCountLimitedBy` – Lets you determine if the result will contain more items than you can use.

See “Determining if a Result Will Return Too Many Items” on page 168.

Determining Whether a Query Returned No Results

Sometimes you write queries that return no results under certain conditions. This often occurs with queries that include predicates that users provide through the user interface. For example, the user interface lets users search for people by name. A user wants to look up someone whose name is “John Smith.” The database could have several matches, or it could have none.

Your user interface displays a list of names when there is a match or the message “No one found with that name” when there is none. Use the `Empty` property on a result object to determine whether a query returned any results.

```
result = query.select()  
if (result.Empty) {
```

As an alternative, you could iterate a result object inside a `while` loop with a counter and test the counter for zero afterwards. Or, you might test the `Count` property against zero. However, relational query performance often improves when you use the `Empty` property.

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 query = Query.make(User)  
query.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 query.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.

Returning Only Unique Items

```
var uniquePerson = query.select().AtMostOneRow
```

Accessing the First Item in a Result

Sometimes you want only the first item in a result, regardless how many instances the database fetched. Use the `FirstResult` property on a result object to obtain its first item.

```
firstPerson = query.select().FirstResult
```

As an alternative, you can iterate a result and stop after retrieving the first item. However, relational query performance often improves when you use the `FirstResult` property to access the only the first item in a result.

Determining if a Result Will Return Too Many Items

When you develop search pages for the user interface, you may want to limit the number of results that you display in the list view. For example, if a user provides little or no search criteria, the number of items returned could be overwhelming. The `getCountLimitedBy` method on result objects lets you efficiently determine how many items a result will return, without fetching all the data from the database. If it will return too many items, your search page can prompt the user to narrow the selection by providing more specific criteria.

With the `getCountLimitedBy` method, you specify a *threshold*. The threshold is the number of items that is too many. In other words, the threshold is the maximum number of items that you want, plus one. If the number of items that the result will contain falls below the threshold, the method returns the actual result count. On the other hand, if the number of items is at or above the threshold, the method returns the threshold, not the actual result count.

For example, you want to limit search results to three hundred items. Pass the number 301 to the `getCountLimitedBy` method, and check the value that the method returns. If the method returns a number less than 301, the result will be within your upper bound, so you can safely iterate the result. If the method returns 301, the result will cross the threshold. In this case, prompt the user to provide more precise search criteria to narrow the result.

The following Gosu sample code demonstrates how to use the `getCountLimitedBy` method.

```
uses gw.api.database.Query

// -- query for addresses in the city of Chicago
var query = Query.make(Address)
query.compare("City", Equals, "Chicago")
var result = query.select()

// -- specify the threshold for too many items in the result --
var threshold = 301 // -- 301 or higher is too many
result.getCountLimitedBy(threshold)

// -- test whether the result count crosses the threshold --
if (result.getCountLimitedBy(threshold) < threshold) {

    // -- iterate the results --
    for (Address in result) {
        print (address.AddressLine1 + ", " + address.City + ", " + address.PostalCode)
    }
}

else {
```

```
// -- prompt for more criteria --
print ("The search will return too many items!")
}
```

Converting Result Objects to Lists, Arrays, Collections, and Sets

You can convert query result sets to lists, arrays, 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 production conditions.

WARNING Converting a query result to a list or array pulls all the entities into local memory. Do this *only* if you are absolutely certain that 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 query = 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 (query.select().Count < 100) {

    var resultCollection = query.select().toCollection()
    var resultSet = query.select().toSet()
    var resultList = query.select().toList()
    var resultArray = query.select().toTypedArray()

    print("Collection: statictypeof " + statictypeof resultCollection + "/ typeof " +
        typeof resultCollection + "/ size " + resultCollection.Count)
    print("Set: statictypeof " + statictypeof resultSet + "/ typeof " +
        typeof resultSet + "/ size " + resultSet.Count)
    print("List: statictypeof " + statictypeof resultList + "/ typeof " +
        typeof resultList + "/ size " + resultList.Count)
    print("Array: statictypeof " + statictypeof resultArray + "/ typeof " +
        typeof resultArray + "/ size " + resultArray.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.

Updating Entity Instances in Query Results

Entities that you iterate across in a query result are read-only by default. The query builder APIs load iterated entities into a read-only *bundle*. A bundle is a collection of entities loaded from the database into server memory that represents a transactional unit of information. The read-only bundles that contain iterated query results are separate from the active read-write bundles of running code. You cannot update the properties of query result entities while they remain in the read-only bundle of the result.

Moving Entities from Query Results to Writable Bundles

To change the properties of entities in query results, you must move the entities from the query result to a writable bundle. To move an entity to a writable bundle, call the `add` method on the writable bundle and save the result of the `add` method in a variable. Whenever you pass an entity from a read-only bundle to a writable bundle, the `add` method returns a clone of the entity instance that you passed to the method.

If you move an entity from a query result to a writable bundle, store the entity reference that the `add` method returns in a variable. Then, modify the properties on the saved entity reference. Do not modify properties on the original entity reference that remains in the result set or its iterator. Avoid keeping any references to the original entity instance whenever possible.

Moving Entities from Query Results to the Current Bundle

Most programming contexts have a writable bundle that the application prepares and manages. For example, all rule execution contexts and PCF code has a current bundle. When a current bundle exists, get it by using the `getCurrent` method.

Typically, whenever you want to update an entity in a query result, you move it to the current bundle. While you iterate the result, add each entity to the current bundle and make changes to the version of it that the `add` method returns. After you finish iterating the query result and the execution context finishes, the application commits all the changes that you made to the database.

IMPORTANT Entities must not exist in more than one writable bundle at a time.

For example:

```
// Get the current bundle from the programming context.  
var bundle = gw.transaction.Transaction.getCurrent()  
  
var query = gw.api.database.Query.make(Address)  
query.compare("State", Equals, typekey.State.TC_IL)  
var result = query.select().orderBy(\row-> row.City)  
  
for (Address in result) {  
    if (!exists (city in {"Schaumburg", "Melrose Park", "Norridge"} where address.City == city)) {  
        continue  
    }  
    else {  
        // Add the address to the current bundle to make it writeable and  
        // discard the read-only reference obtained from the query result.  
        address = bundle.add(address)  
  
        // Change properties on the writeable address.  
        address.Description =  
            // Use a Gosu string template to concatenate the current description and the new text.  
            "${address.Description}; This city is not Schaumburg, Melrose Park, or Norridge.."  
    }  
}
```

At the time the execution context for the preceding code finishes, the application commits all changes made to addresses in the current bundle to the database.

To run the preceding code in the Gosu Scratchpad, add the statement `bundle.commit()` after the `for` block. To see the effects of the preceding code, run the following Gosu code.

```
// Query the database for addresses in Illinois.  
var query = gw.api.database.Query.make(Address)  
query.compare("State", Equals, typekey.State.TC_IL)  
  
// Configure the result to be ordered by City.  
var result = query.select()  
result.orderBy(\ row -> row.City)  
  
// Iterate and print the result set.  
for (Address in result) {  
    print(address.City + ", " + address.Description)  
}
```

See also

“Bundles and Database Transactions” on page 331

Testing and Optimizing Queries

This topic includes:

- “Performance Differences Between Entity and Row Queries” on page 171
- “Viewing the SQL Select Statement for a Query” on page 172
- “Enabling Context Comments in Queries on SQL Server or DB2” on page 173
- “Including Retired Entities in Query Results” on page 174
- “Setting the Page Size for Prefetching Query Results” on page 174
- “Chaining Query Builder Methods” on page 174
- “Working with Advanced Inline Views” on page 175

Performance Differences Between Entity and Row Queries

The query builder APIs support two types of queries:

- **Entity queries** – Result objects contain a list of references to instances of the primary entity type for the query. You set up an entity query with the `select` method that takes no arguments. For example:
`query.select()`
- **Row queries** – Result objects contain a set of row-like structures with values fetched from or computed by the relational database. You set up a row query with the `select` method that takes a Gosu block of column selections. For example:
`query.select(columnSelectionBlock)`

Some situations require entity queries. For example in page configuration, you must use entity queries as the data sources for list views and detail panels. Other situations require row queries. For example, you must use row queries to produce the results of SQL aggregate queries or outer joins. In yet other cases, you can use either type of query. For example, you can use either entity or row queries in Gosu code that runs in batch processes.

Use Entity Queries for Easier Code Understanding and Maintenance

Generally, use an entity query unless performance proves to be an issue. The query builder code for entity queries is easier to understand and to maintain than for row queries. Easier understanding and maintenance is especially true with entity queries if you access the query builder APIs from Java instead of Gosu.

Use Row Queries to Overcome Performance Issue with Entity Queries

Generally, use a row query instead of an entity query if performance issues with an entity query arise. Entity queries sometimes fetch unused data or execute additional queries whenever you navigate object path expressions in the result. Row queries can improve performance in these cases. Row queries fetch only the data you need. Like entity queries, row queries let you join tables to the primary entity so you can include data from related tables in the result.

Performance Issues with Entity Queries

Performance issues with entity queries can occur with the code that processes the results. Navigating object path expressions that span arrays and foreign keys to get to the data that you need can cause additional, implicit database queries to fetch the data. Row queries generally avoid these additional queries.

Improving Performance with Row Queries that Fetch Only the Data that You Need

For entity queries that suffer performance issues, like fetching unused data from the database or navigating object path expression in results, row queries can be a better choice. Row queries retrieve only the data that you need from the database. And, row queries do not cause additional queries to execute at the time you get data from the result.

Row Queries Can Select Entity Instances

The following row query improves performance problems suffered by the previous example of an entity query. The query returns only the necessary entity instances for charges and producers rather than the entire object graphs of invoice items.

Row Queries Can Select Fields

You can reduce the number of queries implicitly executed by row queries if you select specific fields rather than entity instances. If you specify fields rather than instances, the relational database fetches and computes values in response to the SQL query that the query builder expression submits to the database.

For example, the following example improves performance of the preceding row query. Instead of selecting Charge and Producer entity types, the following example selects specific fields on the entity types.

See also

“Limitations of Row Queries” on page 158

Viewing the SQL Select Statement for a Query

The query builder APIs provide two ways to preview and record SQL SELECT statements that your Gosu code and the query builder APIs submit to the application database:

- `toString` – provides an approximation of the SQL Select statement before it is submitted
- `withLogSQL` – displays and records the exact SQL Select statement at the time it is submitted

Using `toString` to Preview SQL Select Statements

You may want to see what the underlying SQL SELECT statement looks like as you build up your query. Use the Gosu `toString` method to return an approximation of the SQL SELECT statement at given points in your Gosu code. That way you can learn how different query builder methods affect the underlying SQL SELECT statement.

For example, the following Gosu sample code prints the SQL Select statement as it exists after creating a query.

```
uses gw.api.database.Query

var query = Query.make(Contact)
print(query.toString())
```

The output looks like the following:

```
[  
    SELECT FROM cc_contact gRoot WHERE gRoot.Retired = 0
```

The first line shows square brackets ([]) with list of variables to bind to the query. In this case, there are none. The remaining lines show the SQL statement. Note that the table for the entity type, cc_contact, has the table alias gRoot.

After you join a related entity to a query or apply a predicate, use the `toString` method to see what the query builder APIs added to the underlying SQL statement.

Note: The `toString` method returns only approximations of SQL statements to be submitted to the application database. SQL statements actually submitted might differ due to database optimizations applied internally by ClaimCenter.

Using withLogSQL to Record SQL Select Statements

You may want to see what the underlying SQL SELECT statement looks like when the query builder APIs submits it to the application database. Use the `withLogSQL` method on a query to see it. When you turn logging behavior on, the query builder APIs write the SQL SELECT statement to the system logs, in logging category `Server.Database`. They also write the SQL statement to standard output (`stdout`).

For example, the following Gosu sample code turns logging on at line 5. The SQL statement is written to the system logs at line 8.

```
1  uses gw.api.database.Query  
2  
3  var query = Query.make(Person)  
4  query.startsWith("LastName", "A", false)  
5  query.withLogSQL(true)      // -- turn logging behavior on here --  
6  var result = query.select()  
7  
8  i = result.iterator()      // -- write the SQL statement to the system logs here --  
9  while (i.hasNext()) {  
10    var person = i.next()  
11    print (person.LastName + ", " + person.FirstName + ":" + person.EmailAddress1)  
12 }
```

The SQL Select statement for the preceding example looks like the following on standard output.

```
Executing sql = Values=[2 (typekey), 4 (typekey), 7 (typekey), 9 (typekey), A% (lastname)]  
SQL=SELECT gRoot.ID col0, gRoot.Subtype col1, gRoot.LastName col2, gRoot.FirstName col3  
FROM pc_contact gRoot  
WHERE gRoot.Subtype IN (?, ?, ?, ?) AND gRoot.LastName LIKE ? ESCAPE '\' AND gRoot.Retired = 0  
ORDER BY col2 ASC, col3 ASC
```

The statement on your system may vary from the preceding example, depending on your relational database.

Note: Writing to the system logs and to standard output does not occur when Gosu sample code calls the `withLogSQL` method. That occurs sometime later, when ClaimCenter submits the query to the relational database.

Enabling Context Comments in Queries on SQL Server or DB2

On SQL Server or DB2, tuning queries can be difficult because you cannot determine what part of the application generates specific queries. To help tune certain queries, enable configuration parameter `IdentifyQueriesViaComments` in `config.xml`.

If you set parameter `IdentifyQueriesViaComments` to true, ClaimCenter adds SQL comments with contextual information to certain SQL SELECT statements that it sends to the relational database. The default for the `IdentifyQueriesViaComments` parameter is false.

The SQL comments are in the format:

```
/* applicationName:ProfilerEntryPoint */
```

The `applicationName` component of the comment is `ClaimCenter`.

The `ProfilerEntryPoint` component of the comment is the name of an entry point known to the Guidewire profiler for that application. For example, `ProfilerEntryPoint` might have the value `WebReq:ClaimSearch`.

Including Retired Entities in Query Results

When an entity instance is retired, its `Retired` property is set to `true`. The *Data Dictionary* tells you which entity types can be retired by including `Retireable` in their lists of delegates. By default, query results do not include retired instances, even if they satisfy all the predicates of the query. For query purposes generally, you must treat retired entities as if they were deleted and no longer in the application database.

To include retired instances in the results of a query, set the `FindRetired` property on the query to `true`, as the following Gosu example code shows.

```
uses gw.api.database.Query

var query = Query.make(User)
query.compare("JobTitle", Equals, "Adjustor")
query.FindRetired = true
```

IMPORTANT Use the `FindRetired` property on queries only under exceptional circumstances that require exposing retired entities.

Setting the Page Size for Prefetching Query Results

When you first begin to iterate a query result, the query builder APIs submit the query to the database. Gosu does not typically submit the query each time you access the next result in the result object. Instead, Gosu automatically loads several results from the database result set as batch into a cache in the application server for quick access. This means that common actions like iterating across the query are higher performance.

In SQL, this is known as *prefetching* results. The number of items that the database prefetches and gives to an application is known as the *prefetch page size*.

You can customize the number of results that the query builder APIs prefetch in order to tune overall query performance from the point-of-view the application server. To set the page size, call the `setPageSize` method on the query result object.

For example:

```
var query = Query.make(User)

// -- prefetch 10 entity instances at one time --
query.select().setPageSize(10)
```

Other notes:

- If you plan to modify the entities, see related section “Updating Entity Instances in Query Results” on page 170.
- In production code, you must not 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 production conditions before and after changing any performance tuning settings.

Chaining Query Builder Methods

You can use a coding pattern known as *method chaining* to write query builder code. With method chaining, the object returned by one method becomes the object from which you call the next method in the chain. Method chaining lets you fit all of your query builder code on a single line as a single statement.

The following sample Gosu code places each query builder API call in a separate statement.

```
uses gw.api.database.Query

// Three query builder APIs in three separate statements.
var query = Query.make(Contact)
query.compare("AddressBookUID", EQUALS, addressBookUID)

var result = query.select()
```

The following sample Gosu code is functionally equivalent to the sample code above, but it uses method chaining to condense three statements into one.

```
uses gw.api.database.Query

// Three query builder APIs chained in a single statement.
var result = Query.make(Contact).compare("AddressBookUID", EQUALS, addressBookUID).select()
```

When you chain methods, the object that supports the chain often does not appear explicitly in your source code. In the example above, the `Query.make` method returns a `Query` object, on which the statement calls the `compare` method. In turn, the `compare` method returns the same `Query` object, on which the statement calls the `select` method. After the statement completes, the `Query` object is discarded and is no longer accessible to subsequent Gosu code.

The reason that you chain the methods of `query`, `table`, and `restriction` objects is they are all the same abstract type and they return themselves.

Method chaining with the query builder APIs is especially useful for user interface development. Page configuration format (PCF) files for list view panels have a single-line property named `value`, which can be a query builder result object.

Working with Advanced Inline Views

You can optionally create what SQL calls an in-line view of these extra columns on the join table. This is an advanced feature which is not normally needed in production code. Use this feature only if necessary due to the performance implications. If you have questions about this, contact Guidewire Customer Support.

In-line views allow you to 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 on-line view, call the `inlineView` method on the query. It returns a query with the new in-line 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 in-line 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 in-line 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`'s related to this `TestE`. This produces the following SQL statement:

```
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.

Method and Type Reference for the Query Builder APIs

Type	Method	Description
Query	compare	Adds a restriction to the query
	join	Returns a table object with information from several entity types joined together in advanced ways.
	having	Returns a restriction object
	select	Returns a result object, with items fetched from a single, root entity.
	subselect	Returns a table object with information from several entity types joined together in basic ways
Table	compare	Adds a restriction to the query
	join	Returns a table object with information from several entity types joined together in advanced ways.
	select	Returns a table object
Restriction	getColumnRef	Represents a boolean condition or conditions on the set of items fetched from the application database.

Method	On class	Description	Returned object type
make	Query	Static method that creates new query objects	Query
select	Query	Fetches items from the application database, according to restrictions added to the query	(IqueryBeanResult) Result
	Table	Fetches items from a specific table that participates in a query	
	Restriction	Fetches items that match a particular restriction within a query	
between	Restriction	Compares a field against a range of values	Restriction
having	Restriction	Compares grouped results to values	Restriction
compare, compareIn, compareNotIn	Query Restriction	Compares a field against a specific value or set of values	Restriction
subselect	Restriction	Joins a secondary source query.subselect(Account.ACCHOLDERCONTACT_PROP.get(), InOperation.CompareIn, Contact.ID_PROP.get())	Table
getColumnRef	Restriction	Creates a reference to an entity field in terms of the database column that stores its values	ColumnRef (does not support chaining)
withDistinct	Restriction	Whether to remove duplicate entity instances from the result	Query
join	Query	Returns a table object with information from several entity types joined together in advanced ways.	Table
startsWith	Query	Compares the beginning values of a character field with a string of characters	Restriction

Method	On class	Description	Returned object type
contains	Query	Compares the value of a character field with a string of characters	Restriction
union	Query	Combines all results from two queries into a single result	GroupingQuery
intersection	Query	Combines results common to two results into a single result.	GroupingQuery
orderBy, orderByDescending, thenBy, thenByDescending	Result	Sort order disregards the locales configured in your ClaimCenter instance. Use the sortBy and sortByDescending methods for locale-specific ordering.	Result

Find Expressions

A Gosu `find` expression lets you find entity instances in the database and then iterate the found results.

IMPORTANT Although `find` expressions are supported, the query builder APIs provide an alternative that supports more fully the features of SQL Select statements. For new code, Guidewire **strongly recommends** that you use the newer query builder APIs rather than `find` expressions. For more information, see “Query Builder APIs” on page 125.

This topic includes:

- “Basic Find Expressions” on page 179
- “Using Exists Expressions for Array Properties in Find Expressions” on page 183
- “Find Expressions that Use Special Substring Keywords” on page 184
- “Using the Results of Find Expressions (Using Query Objects)” on page 185

Basic Find Expressions

Gosu `find` expressions let you query the database using the existing product data model, including data model extensions. You can write useful queries concisely in a single Gosu statement with `find` expression.

A `find` expression uses the following syntax:

```
var queryInstance = find ([var] findIdentifier in findPathExpression where findCriteriaExpression)
```

A `find` expression has the following parts:

- **Identifier** – The symbol you choose for *findIdentifier* serves as shorthand for the entity type that is specified by *findPathExpression*.
- **In clause** – The `in` clause begins with the `in` keyword, followed by *findPathExpression*, which specifies the entity type to find. Find path expressions begin with an entity type literal end, and they end with an entity type literal, not a property. The entity type literal at the end of the path expression sets the type of instances that the expression finds.

- **Where clause** – The where clause begins with the `where` keyword, followed by `findCriteriaExpression`, which contains one or more conditional expressions. The left-hand operands are object path expressions that begin with `findIdentifier` and are relative to the entity type specified by `findPathExpression`. You can use the operators `==`, `!=`, `>`, `<`, `>=`, and `<=`. You can group the conditional expressions with the logical operators `or` and `and`.

The following sample Gosu code finds all activities that are approved. The find path expression begins and ends with the `Activity` entity type, so the found instances are activities. The identifier `a` is shorthand for `Activity` in the object path expression in the where clause. This example queries a direct property of `Activity` instances, the `approved` property.

```
var findQuery = find (activity in Activity where activity.approved) // Find approved activities.
```

The following sample Gosu code finds all activities assigned to external users. This example queries an indirect property of `Activity` instances, the `User.ExternalUser` property that you access through the `AssignedUser` foreign key property on `Activity`.

```
var findQuery = find (activity in Activity where // Find activities assigned to external users.  
                      activity.AssignedUser.ExternalUser  
                    )
```

At runtime, `find` expressions evaluate to `Query` objects. A `find` expression does not submit SQL queries to the database until your code attempts to access information about or retrieve items from the query object.

See also

- For more information, see “Using the Results of Find Expressions (Using Query Objects)” on page 185.
- To learn how to find entity instances based on property values on instances in arrays, “Using Exists Expressions for Array Properties in Find Expressions” on page 183.

Find Expressions that Use AND/OR Operators

Logical AND or OR expressions in a `where` clause are fundamentally the same as those used elsewhere in Gosu.

The following sample Gosu code includes claims with loss types of AUTO while excluding claims with accident types 46 by using the AND operator.

```
var q = find (c in Claim where c.LossType == "AUTO" AND c.AccidentType != "46")
```

The following example uses the OR operator to expands the criteria from the previous example. The expanded criteria include claims with loss types that are null in addition to claims with loss types of AUTO but not with accident types of 46.

```
var q = find (c in Claim where c.LossType == null OR c.LossType == "AUTO" AND c.AccidentType != "46" )
```

The preceding example includes the same items as first example does, but adds to the result claims with null loss types regardless of accident type. The AND operator has precedence over the OR operator. To change the criteria so that Gosu evaluates the OR operator first, use parenthesis to specify the order of evaluation.

```
var q = find (c in Claim where (c.LossType == null OR c.LossType == "AUTO") AND  
                      c.AccidentType != "46"  
                    )
```

Find Expressions that Use Equality and Relational Operators

Gosu supports the following equality and relational operators in the where clauses of `find` expressions:

Equality expressions

`==`

`!=`

Relational expressions

`<`

`>`

`<=`

`>=`

Equality and relational expressions in where clauses are fundamentally the same as those used elsewhere in Gosu. However, there are two restrictions that differentiate these conditional expressions if used in a where clause:

- **Left-hand operand must be a relative query path expression** – The left-hand side of the operator must be an object path expression that begins with the identifier of the `find` expression. 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.

Note: 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` expression. For more information, see “Using Exists Expressions for Array Properties in Find Expressions” on page 183.

- **Right-hand operand must not include a query path expression** – The right-hand side of the operator must not reference a `find` path expression. This means that conditional expressions cannot compare query properties dynamically. All other Gosu expressions are legal on the right-hand side.

There are some subtle distinctions:

- The expression `Claim.LossDate == Exposure.AssignmentDate` is invalid because `Exposure.AssignmentDate` is a query path expression and not an object path expression.
- The expression `Claim.LossDate == exposure.AssignmentDate` is valid, 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.

Note: You cannot reference virtual properties or methods of entity types in a `find` expression.

Because a `find` expression is a high-level abstraction for generating database SQL queries, *virtual* properties or method calls are not supported 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*.

See also

- “Equality Expressions” on page 68
- “Relational Expressions” on page 76

- “Properties” on page 193

Find Expressions that Use 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, sets, and `String` objects (strings are treated as arrays of characters).

Find Expressions and Non-Equality Comparisons

Sometimes database columns are defined as not required. Columns defined this way can contain null values. In Guidewire data model metadata files, columns that allow null values include the attribute `nullOk` in the `column` elements that define them. In the *Data Dictionary*, columns that allow null values are absent the notation `(non-null)`.

SQL Queries and Non-Equality Comparisons of Null Values

SQL queries by default ignore null values if you use the non-equality operator in comparisons with columns that allow null values. Generally in SQL, null values are meant to represent missing, unknown data about which no assertions can be made. In SQL, an assertion that a null value is not equal to some other value is invalid.

For example, you want to find activities based on their end dates, an attribute that allows null values. You want to query the database for activities with end dates that do not fall on 2012-03-15. In a SQL query, activities where end dates are null are not fetched from the database. Only activities where the end date is known and the known end date does not fall on 2012-03-15 are fetched.

Find Expressions and Non-Equality Comparisons of Null Values

In contrast with SQL queries, `find` expressions *do* include null values if you use the non-equality operator in comparison with columns that allow null values. The following sample Gosu code uses a `find` expression finds activities with due dates that do not fall on 2012-03-15. The `find` expression uses the Gosu inequality operator `<>` to compare end dates with “2012-03-15.”

```
var query = find(a in Activity where a.EndDate <> "2012-03-15" as java.util.Date)

for (activity in query) {
    print ("End Date: " + activity.EndDate + "-" + activity.DisplayName)
}
```

The output might look like the following:

```
End Date: 2012-01-27-A new audit has been assigned
End Date: null-A new audit has been assigned
End Date: 2012-03-16-A new audit has been assigned
```

The `find` query result includes the activity with a null end date because Gosu appends the following condition to the SQL `where` clause that it sends to the database.

```
OR EndDate IS NULL
```

To exclude null values from `find` query results, include an explicit condition in the `where` clause, as the following sample expression demonstrates.

```
find(a in Activity where a.EndDate <> "2012-03-15" as java.util.Date and a.EndDate <> null)
for (activity in query) {
    print ("End Date: " + activity.EndDate + "-" + activity.DisplayName)
}
```

The output of from the modified `find` expression above looks like the following:

```
End Date: 2012-01-27-A new audit has been assigned
End Date: 2012-03-16-A new audit has been assigned
```

The `find` query result does not include activities with null end dates.

Converting Non-Equality Comparisons from Find Expressions to Query Builder APIs

The query builder APIs handle non-equality operators in the same way as SQL queries. They both ignore null values. When you convert a legacy `find` expression with a non-equality comparison to a query builder expression, you must add a comparison predicate to include null values. Otherwise, your legacy `find` expression and your new query builder expression behave differently.

The following sample `find` expression finds activities with due dates that do not fall on 2012-03-15.

```
find(a in Activity where a.EndDate <> "2012-03-15" as java.util.Date)
```

The `find` query includes activities with null end dates. To produce the same results with the query builder APIs, you must use the `or` method to include an explicit predicate that includes activities with null end dates.

```
Query.make(Activity).or( \ orCriteria ->
    {orCriteria.compare("EndDate", NotEquals, "2012-03-15" as java.util.Date)
     {orCriteria.compare("EndDate", Equals, null)
    }).select()
```

See also

- “Query Builder APIs” on page 125

Using Exists Expressions for Array Properties in Find Expressions

Use `exists` expressions in the context of `find` expressions to test property values on entity instances in array properties.

An `exists` expression in the context of a `find` expression uses the following syntax:

```
... where ... exists (existsIdentifier in existsPathExpression where existsCriteriaExpression)
```

An `exists` expression in the context of a `find` expression has the following parts:

- Identifier** – A symbol to serve as shorthand for the array property to query, as specified by `existsPathExpression`.
- In clause** – The `in` clause begins with the `in` keyword, followed by `existsPathExpression`, which specifies the array property to query. Exist path expressions in the context of find expressions begin with `findIdentifier` and end with a single value property on the entity type of instances in an array property.
- Where clause** – The `where` clause begins with the `where` keyword, followed by `existsCriteriaExpression`, which contains one or more conditional expressions. The left-hand operands are object path expressions that begin with `existsIdentifier` and are relative the property specified `queryPathExpression`. You can use the operators `==`, `!=`, `>`, `<`, `>=`, and `<=`. You can group the conditional expressions with the logical operators `or` and `and`.

The following sample Gosu code finds claims where the assigned group on an exposure is an Auto Fast Track type of group. The `find` expression uses an `exists` clause to test values on instances in the `Exposures` array property of `Claim` instances.

```
find (claim in Claim where claim.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 SQL queries, including joins if needed.

Generally, you do not need to worry about what SQL is generated for your `find` expressions. However, some `find` expressions that use multiple `exists` expressions may have large performance considerations. Guidewire recommends you consider both the frequency and context of `find` expression 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 `find` expressions as necessary. Also, consider replacing your complex `find` expressions with query builder expressions. The query builder APIs offer more complete support of complex SQL syntax than `find` expressions.

See also

- For uses of an `exists` expression unrelated to `find` expressions, see “Existence Testing Expressions” on page 70.
- “Query Builder APIs” on page 125.

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 the criteria in a `find` expression traverses 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 sample Gosu code finds all claims with a loss type of "AUTO" and has any exposures with an assigned group of type `autofasttrack`.

```
find( c in Claim where c.LossType == "AUTO" and
      exists( group in Claim.Exposures.AssignedGroup where group.GroupType == "autofasttrack" )
    )
```

Combining Exists Expressions

A `find` expression may nest and combine `exists` expressions to any level desired. In other words, `exists` expressions can contain one or more other `exists` expressions, and the outer `find` expression can contain more than one `exists` expression.

Find Expressions that Use Special Substring Keywords

You can use special substring keywords in `find` expressions to search on portions of string values. To search on the initial characters of a string value, use the special keyword `startswith`. To search on characters anywhere within a string value, use the special keyword `contains`.

The following sample Gosu code uses the keyword `startswith` and finds groups with the names `Academy`, `Partners`, and `Acme Insurance`, among others.

```
var query = find( group in Group where group.Name startswith "Ac")
```

The following sample Gosu code uses the keyword `contains` and finds groups with the names Evergreen Indemnity and Greenbay Insurance, among others.

```
var query = find (group in Group where group.Name contains "Green")
```

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 specified by the query path expression. For instance, if you try to find `Contact` instances with a `find` expression, 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)
}
```

Handling Large Query Objects from Find Expressions

If there are concerns about producing an excessively large `Query` object from a `find` expression, 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."
    }
}
```

Sorting Results

You can set the sort order of items in Query objects using the following methods:

- `addAscendingSortColumn`
- `addDescendingSortColumn`

The following example sorts claims by claim number.

```
var query = find (c in Claim where c.ClaimNumber == "54-123456")
query.addAscendingSortColumn("Claim.ClaimNumber")
for (claim in query) {
    print(claim.ClaimNumber)
}
```

Retrieving a Single Row from Find Expression Results

You can use the `getAtMostOneRow` method to retrieve a single item from the Query object that `find` expressions return.

- 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 the `getAtMostOneRow` method only if you are sure that your `find` expression returns at most 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 also

- For details on how to write Gosu classes, see “[Classes](#)” on page 189.
- For more information on using `try...catch...` blocks, see “[Exception Handling](#)” on page 101.

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 running 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
}
```

See also

- “Bundles and Database Transactions” on page 331

Query Objects Returned by Find Expressions 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.

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 100 in the *Configuration Guide*.

Gosu classes are the foundation for syntax of 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.

This topic includes:

- “What Are Classes?” on page 189
- “Creating and Instantiating Classes” on page 190
- “Properties” on page 193
- “Modifiers” on page 198
- “Inner Classes” on page 205

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 built-in 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 204.

For details on using Studio to create and manage your Gosu classes, see “ClaimCenter Studio and Gosu” on page 97 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 239 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. 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("I just created a Tree object!")
    }
}
```

If desired, you can delete the class constructor if you do not need it.

Your class might extend another class. If so, it is often appropriate for your constructor to call its superclass constructor. To do this, use the `super` keyword, followed by parentheses. This statement must be the first statement in the subclass constructor. For example

```
class Tree extends Plant
{
    construct()
    {
        super()
        print("I just created a Tree object!")
    }
}
```

If you call `super()` with no arguments between the parentheses, Gosu calls the superclass no-argument constructor.

If you call `super(parameter_list)`, Gosu calls the superclass constructor that matches the matching parameter list. You can call a superclass constructor with different number of arguments or different types than the current constructor.

For every constructor you write, Gosu always calls some version of the superclass constructor, even if you omit an explicit reference to `super()`. If your class does not call the superclass constructor explicitly:

- If a no-argument constructor exists in the supertype, Gosu calls it, equivalent to the code `super()`.
- If a no-argument constructor does not exist, it is a compilation syntax error. (“no default 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 204.

Although Gosu supports public variables for compatibility with other languages, it is best to always use *public properties backed by private variables* instead of using public variables.

In other words, in your new Gosu classes use this style of variable declaration:

```
private var _firstName : String as FirstName
```

Do not do this:

```
public var FirstName : String           // do not do this. Public variables are not standard Gosu style
```

For more information about defining properties, see “Properties” on page 193.

IMPORTANT The standard Gosu style is to use public properties backed by private variables instead of using public variables. Do not use public variables in new Gosu classes. See “Properties” on page 193 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() // Instantiate a new instance of QueueUtils.
```

You can also use object initializers 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" } // Initialize the ClaimID on the new claim.
```

For more information on new expressions and object initializers, see “New Object Expressions” on page 73.

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 204.

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 79. 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.

Use the following standard package naming conventions:

Type of class	Package	Example of fully qualified class name
Classes you define	<code>customername.subpackage</code>	<code>smithco.messaging.QueueUtils</code>
Guidewire Professional Services classes	<code>gwservices.subpackage</code>	<code>gwservices.messaging.QueueUtils</code>

WARNING Classes and enumerations must never use a package name with the prefix “com.guidewire.” or the prefix “gw.” Those package namespaces are for internal classes only. For more information about reserved packages, see “Importing Types and Package Namespaces” on page 79. If you write Gosu enhancements, the naming rules are slightly different (see “Enhancement Naming and Package Conventions” on page 230).

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. Use 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
}
```

The standard Gosu style is to use 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
```

This declares a private variable called `_firstname`, which Gosu exposes as a public property called `FirstName`.

Do not write your classes to look like:

```
public var FirstName : String
```

IMPORTANT The standard Gosu style is to use 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 Act Like Data But They Are Dynamic and Virtual 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.PropertyA.PropertyB.PropertyC
```

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`.

Gosu provides a variant of the period operator that is always explicitly null-safe for both property access and method access. The null-safe period operator has a question mark before it: `?.`

If the value on the left of the `?.` operator is `null`, the expression evaluates to `null`.

For example, the following expression evaluates left-to-right and contains three null-safe property operators:

```
obj?.PropertyA?.PropertyB?.PropertyC
```

Null Safe Method Calls

By default, 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:

```
house.myaction()
```

If `house` 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`.

In contrast, a *null-safe method call* does not throw an exception if the left side of the period character is `null`. Gosu just returns `null` from that expression. In contrast, using the `?.` operator calls the method with null safety:

```
house?.myaction()
```

If `house` is `null`, Gosu does not throw an exception. Gosu simply returns `null` from the expression.

Null-Safe Versions of Other Operators

Gosu provides other null-safe versions of other common operators:

- The null-safe default operator (`?:`). This operator lets you specify an alternate value if the value to the left of the operator is `null`. For example:

```
var displayName = Book.Title ?: "(Unknown Title)" // return "(Unknown Title)" if Book.Title is null
```

- The null-safe index operator (`?[]`). Use this operator with lists and arrays. It returns null if the list or array value is null at run time, rather than throwing an exception. For example:
`var book = bookshelf?[bookNumber] // return null if bookshelf is null`
- The null-safe math operators (`?+, ?-, ?*, ?/, and ?%`). For example:
`var displayName = cost ?* 2 // multiply times 2, or return null if cost is null`

See “Handling Null Values In Expressions” on page 83.

Design Code for Null Safety

Use null-safe operators where appropriate. They make code easy to read and easier to handle edge cases.

You can also design your code to take advantage of this special language feature. For example, expose data as *properties* in Gosu classes and interfaces rather than setter and getter methods.

See Also

- For more examples and discussion, see “Handling Null Values In Expressions” on page 83

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 193

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 the type `Boolean` (the type of the `Empty` property), the expression evaluates to `false` in either of the following cases:

- if `obj.StringProperty` has the value `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 204.

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" )
}
```

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
<code>public</code>	Fully accessible. No restrictions.	Yes	Yes	Yes	Yes	Yes	Yes
<code>protected</code>	Accessible only by types with same package and subtypes.	--	Yes	Yes	Yes	Yes	--
<code>internal</code>	Accessible only in same package	Yes	Yes	Yes	Yes	--	--
<code>private</code>	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	<code>public</code>
Variables	<code>private</code>
Functions	<code>public</code>
Properties	<code>public</code>

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 395.

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 = "ABCDEFG"
print("License plate = " + t.Plate)
t.RegisterWithDMV( "http://dmv.ca.gov/register" )
```

This prints the following:

```
License plate = ABCDEFG
Pretending to register ABCDEFG to http://dmv.ca.gov/register
```

Final Modifier

The `final` modifier applies to types (including classes), type members (variables, properties, methods), local variables, and function parameters.

The `final` modifier 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 209. This means that no Gosu code can subclass an enumeration.

Final Class Variables

The `final` keyword can be used on class variables, which means that the variable can be set only once, and only in the declared class (not by subclasses).

For example:

```
class TestABC {
    final var _name : String = "John"
}
```

Optionally, you can use the `final` keyword on a class variable declaration without immediately initializing the variable. If you do not immediately initialize the variable in the same statement, all class constructors must initialize it in all possible code paths.

For example, the following syntax is now valid because all constructors initialize it once in each code path:

```
class TestABC {
    final var _name : String

    construct() {
        _name = "hello"
    }
    construct(b : boolean){
        _name = "there"
    }
}
```

However, the following is invalid because one constructor does not initialize the final variable:

```
class TestABC {
    final var _name : String // INVALID CODE, ALL CONSTRUCTORS MUST INITIALIZE THIS IN ALL CODE PATHS

    construct() { // does not initialize the variable
    }
    construct(b : boolean){
        _name = "there"
    }
}
```

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 193.

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 local variable as `final`, you can initialize it with a value immediately as you declare the variable.

Optionally, you can declare the variable as `final` but not immediately initialize it with a value. You must set the value eventually in that function for all possible code paths.

For example, you can write something like this:

```
function foo() {
    final var b : int
    b = 10
}
```

The Gosu compiler verifies all code paths have initialization exactly once. In other words, any `if` statements or other flow control structures must set the variable and only once.

For example, the following code is valid:

```
function foo(a : boolean) {
    final var b : int
    if(a) {
        b = 0
    } else {
        b = 1
    }
}
```

However, if you remove the `else` branch it is invalid because as the `final` variable is initialized only if `a` is `true`.

```
function foo(a : boolean) {
    final var b : int // INVALID CODE, UNINITIALIZED IF "a" IS FALSE
    if (a) {
        b = 0
    }
}
```

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 Final
{
    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 369.

For additional help on static variables and synchronized access, contact 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 199.

The `static` modifier cannot be combined with the `abstract` modifier. See “Abstract Modifier” on page 200 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:

```
Bonjour
```

For more information about this topic, refer to the next section, “Inner Classes” on page 205.

Inner Classes

You can define inner classes in Gosu, similar to inner classes in Java. They are useful for encapsulating code even further within 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 239).

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 204.

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( "ABCDEFG" )  
        }  
    }  
}
```

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 ABCDEFG
```

Gosu Block Shortcut for Anonymous Classes Implementing an Interface

In certain cases, you can pass a block as a method argument instead of an instance of an anonymous class. If the method is part of an interface that contains exactly one method, you can pass a block instead of the anonymous class instance. This is especially helpful for APIs defined to take the type `BlockRunnable` or `Runnable`.

For more information, see “Blocks as Shortcuts for Anonymous Classes” on page 237.

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” on page 209

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.")
```

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?” on page 211
- “Defining and Using an Interface” on page 212

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 163 in the *Integration Guide*.

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 looks like a property 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 213.

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 239.)

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 Scratchpad:

```
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 Scratchpad code as before, it prints the same results.

For information about Gosu class properties in general, see “Classes” on page 189.

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 198.

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  
}
```

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 211
- “Classes” on page 189

This topic includes:

- “Using Gosu Composition” on page 215

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 367.

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" )
```

Annotations

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.

This topic includes:

- “Annotating a Class, Method, Type, Class Variable, or Argument” on page 219
- “Annotations at Run Time” on page 222
- “Defining Your Own Annotations” on page 223

Annotating a Class, Method, Type, Class Variable, or Argument

Annotations are a simple syntax to add metadata to a Gosu class, a class member, a function parameter.

For example, annotations could add indicate what a method returns, or indicate what kinds of exceptions the method might throw. You can add custom annotations and read this information at run time.

To add an annotation, type the at sign (@), followed by the annotation name immediately before the declaration of what it annotates.

For example, the following simple example declares a class is deprecated:

```
@Deprecated  
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 that were defined natively in Gosu and you can also directly use Java annotations.

If you do not type the fully-qualified name of the annotation when you use it, add a `uses` line at the top of the file for that annotation class. See “Importing Types and Package Namespaces” on page 79.

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 annotations support the named arguments calling convention:

```
@KnownBreak(:targetUser = "user", :targetBranch = "branch", :jira = "ABC-xxxxx")
```

For related information about named arguments, see “Named Arguments and Argument Defaults” on page 98.

Function Argument Annotations

The Gosu language supports annotations on function parameters, including Gosu block declarations. In some cases you need to explicitly add `uses` lines to declare which annotation class to use.

For example:

```
package test

uses java.lang.Integer
uses java.lang.Deprecated
uses javax.annotation.Nonnull

class TestABC {

    function add (@Nonnull a : Integer, b : Integer) : Integer {
        return a + b
    }

    function addAndLog (@Deprecated a : Integer, b : Integer) : Integer {
        return a + b
    }
}
```

There are currently no compile time APIs for changing IDE behavior based on the annotations.

You can get the function parameter annotations using Java reflection APIs. See “Gosu Class Function Parameter Argument Annotations at Run Time” on page 223.

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 will go away in a future release. Begin rewriting code to avoid using this class, method, constructor, property, or function parameter.	Can appear anywhere, but only once for any specific class, method, constructor, property, or function argument.	(1) A warning string to display if this deprecated class, method, or constructor is used.
@SuppressWarnings	Gosu provides limited support for the Java annotation @SuppressWarnings, which tells the compiler to suppress warnings.	Declarations of a type, function, property, constructor, field, or parameter. Note that local variables do not support this annotation.	You must pass a String value as an argument to indicate what warnings to suppress. Pass the argument "all" to suppress all warnings. Pass the argument "deprecation" to suppress deprecation warnings. For example, to suppress deprecation warnings in a Gosu class, add the @SuppressWarnings("deprecation") on the line before the class declaration.

The following code uses several built-in annotations:

```
package com.mycompany
uses java.lang.Exception

@WsiWebService
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*. A deprecated API is temporarily available but a future release will remove it. Immediately start to refactor code that uses deprecated APIs. This ensures your code is compatible with future releases. Following this advice will simplify future upgrades.

```
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 223), 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 {
    ...
}
```

Internal Annotations

There are 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 these internal annotations so that you understand their role in built-in classes.

Internal annotation	Description	Usage limits
<code>@Export</code>	Let a class be visible and editable in Studio	Classes only
<code>@ReadOnly</code>	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

Web Service Annotations

Several built-in annotations related to publishing web services are discussed further in the following sections:

- For WS-I web service publishing, see “Publishing Web Services (WS-I)” on page 37 in the *Integration Guide*
- For RPCE web service publishing, see “Publishing Web Services (RPCE)” on page 89 in the *Integration Guide*

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 <code>@Deprecated</code> annotation
Type	<code>(typeof obj).TypeInfo.getAnnotation(Deprecated)</code>
Constructor	<code>(typeof obj).TypeInfo.Constructors[i].getAnnotation(Deprecated)</code>
Method	<code>(typeof obj).TypeInfo.Methods[i].getAnnotation(Deprecated)</code>
Property	<code>(typeof obj).TypeInfo.Properties[i].getAnnotation(Deprecated)</code>

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 239.

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	<code>(typeof obj).TypeInfo.Annotations</code>
Constructor	<code>(typeof obj).TypeInfo.Constructors[i].Annotations</code>
Method	<code>(typeof obj).TypeInfo.Methods[i].Annotations</code>
Property	<code>(typeof obj).TypeInfo.Properties[i].Annotations</code>

For a detailed example of accessing annotations at run time, see “Defining Your Own Annotations” on page 223.

Gosu Class Function Parameter Argument Annotations at Run Time

The syntax for getting annotations at run time from Gosu has a different calling style than annotations on types, constructors, methods, or properties.

For a Gosu class, first get access to the Java backing class for the Gosu class. Next, use the native Java reflection APIs to access the parameter annotations.

The following example gets the first annotation of the first parameter of a method:

```
// Get the Java backing class for the Gosu class, using reflection (not type safe)
IGosuClass gsClass = (IGosuClass)TypeSystem.getByName( "my.gosu.Sample" );
Class cls = gsClass.getBackingClass();

// Get a reference to the method using reflection (not type safe)
Method fooMethod = cls.getMethod( "foo" );

// Using the native Java API for annotations, get the annotations list
Annotation[][] paramAnnotations = fooMethod.getParameterAnnotations();

// In this artificial example, get the first annotation of the first parameter...
Annotation firstAnnotationOnFirstParameter = paramAnnotations[0][0];
System.out.println( "First annotation on first parameter: " + firstAnnotationOnFirstParameter );
```

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.guidewire.pl.docexamples.annotations` package

```
package com.guidewire.pl.docexamples.annotations  
  
class Author implements IAnnotation {  
}
```

In this case the annotation has no constructor, which implies the annotation takes no parameters. You can call it as:

```
@Author()
```

Because there are no arguments, you can optionally 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 be 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 225.

Test this annotation by using it on a newly defined type, such as a new Gosu class. Create the following class in the `com.guidewire.pl.docexamples.annotations` package:

```
package com.guidewire.pl.docexamples.annotations  
  
uses com.guidewire.pl.docexamples.annotations.Author  
  
@Author("A. C. Clarke")  
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. Call the `Instance` property to get the annotation and cast it to your desired more specific annotation class using the `as` operator. The `Instance` property throws an exception if there is more than one instance of that annotation on the type in that context.

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  
  
var a = Spaceship.Type.TypeInfo.getAnnotation(Author).Instance as Author  
  
print(typeof a)  
print ("The author name from the annotation is ${a.AuthorName}")
```

This example prints the following:

```
com.mycompany.Author  
The author name from the annotation is A. C. Clarke
```

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. Declare with values of the `gw.lang.annotation.UsageTarget` enumeration:

- `MethodTarget` – This annotation can be used on a method
- `TypeTarget` – This annotation can be used on a type, including classes
- `PropertyTarget` – This annotation can be used on a property
- `ConstructorTarget` – This annotation can be used on a constructor
- `ParameterTarget` – This annotation can be used on a function argument (a parameter)
- `AllTarget` – This annotation can be used on all targets listed above

The *modifier* defines how many times the annotation can be used for that target. Declare with values of the `gw.lang.annotation.UsageModifier` enumeration:

- `None` – This annotation cannot exist on that target
- `Once` – This annotation can only appear once on that target
- `Many` – This annotation can appear many (unlimited) times on that target

For example, the built-in annotation called `@Returns` annotation can only appear on methods and can only appear once. Define such requirements right before the class definition with a line such as:

```
@AnnotationUsage(annotation.UsageTarget.Method, annotation.UsageModifier.One)
```

The default availability is universal targets and modifiers. 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.

If you use any `AnnotationUsage` annotation in an annotation definition, all targets default to `None` for any targets not otherwise specified. If you use `AnnotationUsage` once, Gosu requires you to explicitly specify supported targets using `AnnotationUsage` meta-annotations.

Add multiple lines to describe each type of use you want to permit.

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” on page 227

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 231.) Gosu 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 251. If you have not yet learned about Gosu *blocks*, you may want to first review “Gosu Blocks” on page 231.

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: “`: TYPE TO EXTEND`” 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] [OptionalFunctionalDescription]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 198 for more information about the **protected** keyword. However, to avoid namespace conflicts with built-in 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.

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?” on page 231
- “Basic Block Definition and Invocation” on page 232
- “Variable Scope and Capturing Variables In Blocks” on page 234
- “Argument Type Inference Shortcut In Certain Cases” on page 235
- “Block Type Literals” on page 235
- “Blocks and Collections” on page 237
- “Blocks as Shortcuts for Anonymous Classes” on page 237

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 Database Transactions” on page 331.

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 235.

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
```

In general, the standard Gosu style is to omit all semicolon characters in Gosu at the end of lines. Gosu code is more readable without optional semicolons. However, if you provide statement lists on one line, such as within block definitions, use semicolons to separate statements. For other style guidelines, see “General Coding Guidelines” on page 395.

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 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 251.

Block Type Literals

Block literals are a form of type literal, which means the way you reference a *block type*. The block literal specifies 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 251.

Blocks as Shortcuts for Anonymous Classes

In certain cases, you can pass a block as a method argument instead of an instance of an anonymous class. If the method is part of an interface that contains exactly one method, you can pass a block instead of the anonymous class instance. This technique in Gosu is an easy-to-read coding style that works with interfaces that were originally implemented in either Gosu or Java. The parameters of the block must be the same number and type as the parameters to the interface method. The return type of the block must be the same as the return type of that method.

Use a block for APIs that use the Gosu interface type `BlockRunnable` or the Java interface type `Runnable`. Both interfaces are simple containers for a single method called `run`.

For example, suppose a method signature looks like the following:

```
public function doAction(b : BlockRunnable)
```

You can call this method using a block:

```
obj.doAction(\ -> print("do your action here"))
```

As a naming convention, if an API uses a type with a name that contains the word `Block`, then you can probably use a block for that type.

This feature works with any interface, including interfaces defined as inner interfaces within a class. 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 CallbackBlock {  
        public void run() throws Throwable;  
    }  
  
    // ...  
  
    public void execute(CallbackBlock 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 code it more concisely with a block:

```
public function messageReceived(messageId : int) {
    _callbackHandler.execute(\ -> { /* your Gosu statements here */ })
}
```

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 be very general about the type of the contents, such as using a root type such as `Object`. However, if you are 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 classes 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`” with APIs that you design 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 31.

This topic includes:

- “Gosu Generics Overview” on page 240
- “Using Gosu Generics” on page 241
- “Other Unbounded Generics Wildcards” on page 243
- “Generics and Blocks” on page 244
- “How Generics Help Define Collection APIs” on page 246
- “Multiple Dimensionality Generics” on page 247
- “Generics With Custom ‘Containers’” on page 247

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 47 `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.

An alternative is to define different types of collections or lists supporting only homogenous sets of objects of a certain type, such as `StringList`, `IntegerList`, or `MyCustomClassList`. Defining homogenous sets of objects in this way provides compile-time certainty and 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, the method definitions might be incomplete. If you provide such an API, it cannot predict a list of types you do not know about that a consumer of your API wants to use. If 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 that you could define a collection with an explicit type of each item. By using 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 the syntax `List<Number>` means *a list of numbers*. Even better, suppose there were a way to define function parameters to work with **any type**. What if it always returned an object of the same type, or an array of that type, and had 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 confirms that 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 of 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 compati-

bility, which is that Gosu does not support the syntax `<? super TYPE>`. For more information about other similar features, see “Bounded Wildcards” on page 244.

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*—unlike Java, at run time, Gosu retains the actual specific type. For example, 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 a return value by using Gosu generics, a “consumer” of this API finds the API 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 251 and “Basic Hash Maps” on page 253.

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 is even simpler 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.

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 called *parameterization*.

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 c = myCollection.iterator.next()
(c as MyClass).myClassMethod()
```

Note the code `(c 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. For example, suppose that the local variable (in this case, `c`) already carried with it the information about the type of the object in the collection. That allows you to remove the "as `MyClass`" before calling methods on the object:

```
var c = new ArrayList<MyClass>() { c1, c2, c3 }
...
// the result of this is strongly typed
var first = c.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 `CLASSENAME<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. In Gosu this is called *parameterizing* the method. In other languages, this is known as 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 inline within another function (see “Gosu Blocks” on page 231). 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 244.

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. Be aware the *sortBy* method sorts the original list in place, and does not return an entirely new list. If you want to return a new list, use the *orderBy* method instead.

The following example sorts a list of *String* objects by the length of the *String*.

```
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 246.

For extensive information about similar APIs with blocks, see “Gosu Blocks” on page 231. For specific examples of built-in APIs that use generics with blocks, see “Collections” on page 251.

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 251

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 dimensions. See “Multiple Dimensionality Generics” on page 247. You could define your class abstracted across multiple types, separated by commas:

```
public class MyClass<K, V>
```


Collections

Gosu collection and list classes rely on collection classes from the Java language. However, Gosu collections and lists have significant built-in enhancements compared to Java. For example, *Gosu blocks* are anonymous in-line defined functions that the Java language does not support. By using the enhanced Gosu collection and list classes, with a single line of code you can loop through collection items to perform actions, extract item information, or sort items.

This topic includes:

- “Basic Lists” on page 251
- “Basic Hash Maps” on page 253
- “List and Array Expansion (*.*)” on page 255
- “Enhancement Reference for Collections and Related Types” on page 257

See also

- “Gosu Blocks” on page 231
- “Gosu Generics” on page 239
- “Enhancements” on page 227

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 239.

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.

See also

“Compound Types” on page 367

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 highest 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 Interface in Gosu

In new expressions, you can use the interface type `List` rather than the class type `ArrayList` to instantiate a class. 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 Hash Maps

Maps in Gosu inherit from the Java class `java.util.HashMap`.

Creating a Hash Map

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>()
```

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.

See also

"Gosu Generics" on page 239

Getting and Setting Values in a Hash Map

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 Hash Map 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` – Calculates 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) )
```

Wrapped Maps with Default Values

Gosu provides a class called `gw.util.AutoMap` that implements the Java interface `java.util.Map`. This is an alternative to using the standard `java.util.HashMap` class.

The `AutoMap` class wraps a `java.util.Map` and provides a default value for the map if the key is not present. When you create an `AutoMap` object, you pass a Gosu block into the constructor. If some calling code calls `get` on the map and its value is `null`, Gosu runs a mapping block that you provide. Gosu takes the return value from the block, stores it in the map for that key, then returns the value to the original caller. All other methods delegate directly to the wrapped map.

The most simple constructor takes only one argument, which is a Gosu block that provides a default value. The block takes one argument (the key) and returns the default value to use. If you use this constructor, Gosu creates a new instance of `java.util.HashMap` as its wrapped map.

Another constructor takes a map (any object whose class implements `java.util.Map`) as the first argument and the Gosu block as the second argument. Use this alternate constructor if you already have an existing map that you want to wrap rather than create a new map.

For example:

```
var origMap = { 1 -> "apple", 4 -> "orange"}  
var newMap = origMap.toAutoMap( \ k -> "I want ${k} blueberries")  
for (i in 1..5) {  
    print(newMap[i])  
}
```

This code prints:

```
apple  
I want 2 blueberries  
I want 3 blueberries  
orange  
I want 5 blueberries
```

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.

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.

See also

[“Enhancement Reference for Collections and Related Types” on page 257](#)

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 Scratchpad 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

Gosu collection and list classes rely on collection classes from the Java language. However, Gosu collections and lists have significant built-in *enhancements* compared to Java. Gosu enhancements are Gosu methods and/or properties added to classes or other types without requiring subclassing to make use of the new methods and properties. 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,

discussed in “Gosu Generics” on page 239. 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 or property	Description
<code>Count</code>	Returns the number of elements in the <code>Iterable</code>
<code>single()</code>	If there is only one element in the <code>Iterable</code> , that value is returned. Otherwise an <code>IllegalStateException</code> is thrown.
<code>toCollection()</code>	If this <code>Iterable</code> object is already of type <code>Collection</code> , return it. Otherwise, copy all values out of this <code>Iterable</code> into a new <code>Collection</code> .

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 or property	Description
<code>allMatch(cond)</code>	Returns true if all elements in the <code>Collection</code> satisfy the condition
<code>hasMatch(cond)</code>	Returns true if this <code>Collection</code> has any elements in it that match the given block
<code>asIterable()</code>	Returns this <code>Collection<T></code> as a pure <code>Iterable<T></code> (in other words, not as a <code>List<T></code>).
<code>average(selector)</code>	Returns the average of the numeric values selected from the <code>Collection<T></code>
<code>countWhere(cond)</code>	Returns the number of elements in the <code>Collection</code> that match the given condition
<code>HasElements</code>	Returns true if this <code>Collection</code> has any elements in it. This is a better method to use than the default collection method <code>empty()</code> because <code>HasElements</code> interacts better with <code>null</code> values. For example, the expression <code>col.HasElements()</code> returns a non-true value even if the expression <code>col</code> is <code>null</code> .
<code>first()</code>	Returns first element in the <code>Collection</code> , or return <code>null</code> if the collection is empty.
<code>firstWhere(cond)</code>	Returns first element in the <code>Collection</code> that satisfies the condition, or returns <code>null</code> if none do.
<code>flatMap(proj)</code>	Maps each element of the <code>Collection</code> to a <code>Collection</code> of values and then flattens them into a single <code>List</code> .
<code>fold()</code>	Accumulates the values of an <code>Collection<T></code> into a single <code>T</code> .
<code>intersect(iter)</code>	Returns a <code>Set<T></code> that is the intersection of the two <code>Collection</code> objects.
<code>last()</code>	Returns last element in the <code>Collection</code> or return <code>null</code> if the list is empty.
<code>lastWhere(cond)</code>	Returns last element in the <code>Collection</code> that matches the given condition, or <code>null</code> if no elements match it.
<code>map(proj)</code>	Returns a <code>List</code> of each element of the <code>Collection<T></code> mapped to a new value.
<code>max(proj)</code>	Returns maximum of the selected values from <code>Collection<T></code>
<code>min(proj)</code>	Returns minimum of the selected values from <code>Collection<T></code>
<code>orderBy(proj)</code>	Returns a new <code>List<T></code> ordered by a block that you provide. Note that this is different than the <code>sortBy</code> method, which is retained on <code>List<T></code> and which sorts in place. Note: The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface <code>java.lang.Comparable</code> . Because of this, these methods do not sort <code>String</code> values in a locale-sensitive way.
<code>orderByDescending(proj)</code>	Returns a new <code>List<T></code> reverse ordered by the given value. Note that this is different than <code>sortByDescending()</code> , which is retained on <code>List<T></code> and which sorts in place. Note: The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface <code>java.lang.Comparable</code> . Because of this, these methods do not sort <code>String</code> values in a locale-sensitive way.
<code>partition(proj)</code>	Partitions this <code>Collection</code> into a <code>Map</code> of keys to a list of elements in this <code>Collection</code> .
<code>partitionUniquely(proj)</code>	Partitions this <code>Collection</code> into a <code>Map</code> of keys to elements in this <code>Collection</code> . Throws an <code>IllegalStateException</code> if more than one element maps to the same key.
<code>reduce(init, reducer)</code>	Accumulates the values of a <code>Collection<T></code> into a single <code>V</code> given an initial seed value.
<code>reverse()</code>	Reverses the collection as a <code>List</code> .
<code>singleWhere(cond)</code>	If there is only one element in the <code>Collection</code> that matches the given condition, it is returned. Otherwise an <code>IllegalStateException</code> is thrown
<code>sum(proj)</code>	Returns the sum of the numeric values selected from the <code>Collection<T></code>
<code>thenBy(proj)</code>	Additionally orders a <code>List</code> that has already been ordered by <code>orderBy</code> . Note: The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface <code>java.lang.Comparable</code> . Because of this, these methods do not sort <code>String</code> values in a locale-sensitive way.

Method or property	Description
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 <code>java.lang.Comparable</code> . 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 or property	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 <code>java.lang.Comparable</code> . 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 <code>java.lang.Comparable</code> . Because of this, these methods do not sort String values in a locale-sensitive way.

Methods on Map<K,V>

The following table lists the available methods on Set<T>.

The methods that have `retain` in the name are destructive. The methods with `filter` in the name create a new map and do not modify the original map.

Method or property	Description
General usage	
<code>Count</code>	The number of keys in the map.
<code>Keys</code>	A collection of all keys in the map.
<code>Values</code>	A collection of all values in the map.
<code>copy()</code>	Returns a HashMap object that is a copy of this Map. This method always returns a HashMap object, even if the original map was another class that implements the Map interface.
<code>toAutoMap(defaultValueBlock)</code>	Returns an AutoMap object that wraps the current map and provides handling of default values. The method takes a default value mapping block with the same meaning as in the constructor for the AutoMap object. The block takes one argument (the key) and returns a default value that is appropriate for that key. See "Wrapped Maps with Default Values" on page 254.
Perform an action repeatedly	
<code>eachKeyAndValue(block)</code>	Runs a block for every key-value pair in the map. The block must take two arguments: a key and a value. The block must return no value.
<code>eachKey(block)</code>	Runs a block for every key in the map. The block must take one argument (the key) and return no value.
<code>eachValue(block)</code>	Runs a block for every value in the map. The block must take one argument (the value) and return no value.
Filtering and remapping	
<code>retainWhereKeys(keyFilter)</code>	Destructively removes all entries whose keys do not satisfy the <code>keyFilter</code> expression. Returns true if and only if this map changed as a result of the block. The key filter block must take one argument (a key) and return true or false.
<code>retainWhereValues(valueFilter)</code>	Destructively removes all entries whose values do not satisfy the <code>valueFilter</code> expression. Return true if this map changed as a result of the call. The value filter block must take one argument (a value) and return true or false.
<code>filterByKeys(keyFilter)</code>	Returns a new map that is a clone of the original map but without entries whose keys do not satisfy the <code>keyFilter</code> expression. The key filter block must take one argument (a key) and return true or false.
<code>filterByValues(valueFilter)</code>	Returns a new map that is a clone of the original map but without entries whose values do not satisfy the <code>valueFilter</code> expression. The key filter block must take one argument (a value) and return true or false.
<code>mapValues(mapperBlock)</code>	Returns a new map that contains the same keys as the original map but different values based on a block that you provide. The block must take one argument (the original value) and return a new value based on that value. The values can have an entirely different type. Gosu infers the return type of the block based on what you return and creates the appropriate type of map. For example, the following Gosu takes a <code>Map<Integer, String></code> and maps it to a new object of type <code>Map<Integer, Integer></code> .
	<code>var origMap = { 1 -> "hello", 4 -> "there"} var newMap = origMap.mapValues(\ value -> value.length)</code>
Properties files	

Method or property	Description
<code>writeToPropertiesFile(file)</code> <code>writeToPropertiesFile(file, comments)</code>	Writes the contents of this map to a file in the Java properties format using APIs on <code>java.util.Properties</code> . The method takes a <code>java.io.File</code> object and returns nothing. The second method signature takes a <code>String</code> object to write comments in the properties file.
<code>readFromPropertiesFile(file)</code>	Reads the contents of this map from a file in the Java properties format using APIs on <code>java.util.Properties</code> . The method takes a <code>java.io.File</code> object and returns a map of the keys and values. The return type is <code>Map<String, String></code> .

Methods on Set<T>

The following table lists the available methods on `Set<T>`.

Method or property	Description
<code>copy()</code>	Creates a copy of the set
<code>powerSet()</code>	Returns the power set of the set
<code>freeze()</code>	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`.

If `firstWhere` finds no matching items, it returns `null`.

Similarly, there is a `lastWhere` method that finds the last item that matched the condition, and returns `null` if none are found.

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 179.

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.

Be aware the `sortBy` method sorts the original list in place, and does not return an entirely new list. If you want to return a new list, use the `orderBy` method instead.

The following example sorts strings by their length:

```
var resortedStrings = myStrings.sortBy( \ str -> str.Length )
```

If you want to print the contents, print them with:

```
resortedStrings.each( \ str -> print( str ) )
```

This produces 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.

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` of elements from a collection or array that is a property on elements on an outer collection. Suppose you have a collection where each object has a property that is an array or collection. You provide a block that takes the elements of the inner collection and returns them an array or collection. The `flatMap` method concatenates all the elements in the returned collections or arrays into a single `List`.

For example, suppose your data has the following structure:

- 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.

The `Claim.Exposures` property is the outer collection of exposures. The `Exposure.Notes` properties are the inner collections.

First, write a simple block that extracts the note objects from an 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 that contains all the notes on all the exposures on the claim. In generics notation, the `flatMap` method returns an instance of `List<Note>`.

The `flatMap` method is similar to the array expansion feature of Gosu. However, the `flatMap` method is available on all collections and arrays. In addition, the `flatMap` method generates different extracted arrays dynamically using a Gosu block that you provide. Your Gosu block can perform any arbitrary and potentially-complex calculation during the flat mapping process.

See also

[“Array Expansion” on page 53](#)

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.

Gosu and XML

XML files describe complex structured data in a text-based format with strict syntax for easy data interchange. Gosu can read or write any XML document. If you have an associated XSD to define the document structure, Gosu parses the XML using the schema to produce a statically-typed tree of XML elements with structured data. Also during parsing, Gosu can validate the XML against the schema. You can manipulate XML or generate XML without an XSD file, but use XSDs if possible. Without an XSD, your XML elements do not get programming shortcuts, such as Gosu properties on each element, nor intelligent static typing.

This topic includes:

- “Manipulating XML Overview” on page 268
- “Introduction to the XML Element in Gosu” on page 271
- “Exporting XML Data” on page 275
- “Parsing XML Data into an XML Element” on page 278
- “Creating Many Qualified Names in the Same Namespace” on page 280
- “XSD-based Properties and Types” on page 281
- “Getting Data From an XML Element” on page 288
- “Simple Values” on page 291
- “Access the Nullness of an Element” on page 294
- “Automatic Creation of Intermediary Elements” on page 295
- “Default and Fixed Attribute Values” on page 295
- “Substitution Group Hierarchies” on page 296
- “Element Sorting for XSD-based Elements” on page 297
- “Built-in Schemas” on page 300
- “Use a Local XSD for an External Namespace or XSD Location” on page 301
- “Schema Access Type” on page 301
- “The Guidewire XML (GX) Modeler” on page 302
- “Legacy XML APIs: Manipulating XML as Untyped Nodes” on page 313
- “Legacy XML APIs: Exporting XML Data” on page 316

- “Legacy XML APIs: Collection-like Enhancements for XML” on page 318
- “Legacy XML APIs: Structured XML Using XSDs” on page 319

For more information on the Extensible Markup Language (XML), refer to the World Wide Web Consortium web site <http://www.w3.org/XML>.

Manipulating XML Overview

To manipulate XML in Gosu, Gosu creates an in-memory representation of a graph of XML elements. The main Gosu class to handle an XML element is the class called `XmLElement`. Instead of manipulating XML by modifying text data in an XML file, your Gosu code can simply manipulate `XmLElement` objects. You can read in XML data from a file or other sources and parse it into a graph of XML elements. You can export a graph of XML elements as standard XML, for example as an array of bytes containing XML data.

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 (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.

To skip ahead to an overview about the current XML APIs, see “Introduction to the XML Element in Gosu” on page 271.

Legacy XML Support

In previous releases, the Gosu native XML APIs worked differently. This documentation refers to that implementation as the *legacy XML APIs*. The current XML APIs use the base class `XmLElement`. The legacy XML APIs use the class `gw.xml.XMLNode` and the interface `gw.xml.IXMLnode`. The legacy XML APIs continue to work in the current product. However, for all new development, use the current XML APIs not the legacy XML APIs.

IMPORTANT ClaimCenter provides the legacy XML APIs for backward compatibility. However, for any new XML parsing, object manipulation code, or XSD parsing, always use the current APIs that are based on `XmLElement`.

The legacy XML APIs support XML untyped nodes or as statically typed XML nodes using an XSD.

There are two important types that support the legacy XML APIs. 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 using legacy XML APIs:

- “Legacy XML APIs: Manipulating XML as Untyped Nodes” on page 313
- “Legacy XML APIs: Structured XML Using XSDs” on page 319

The legacy XML APIs support fewer advanced features of the XSD specification. For details, see “Limitations of XSD Support” on page 322.

Differences Between Legacy XML APIs and Current XML APIs

The following table summarizes major differences between the legacy XML APIs and the current XML APIs.

Feature	Legacy XML APIs	Current XML APIs
XSD support	Limited XSD feature support	Standards-compliant XSD support
Class representing an XML element or node	XMLNode	XmLElement
Naming convention	built-in types start with XML (capital letters for XML), such as XMLNode	Built-in types use camel-case naming, such as XmLElement
Gosu type for the names of elements and attributes	String	<p>QName. A QName is a qualified name that includes a namespace String and a type name. You can use the following syntax to create a new QName with an empty (<code>null</code>) namespace for a specified localPart:</p> <pre>new QName("ThisIsMyString")</pre>
Gosu type for the values of attributes and simple values	String	<p><code>XmLSimpleValue</code>. An <code>XmLSimpleValue</code> has the following properties:</p> <ul style="list-style-type: none"> • it can use any type of backing storage • it can produce a backing value (for example, <code>int</code>), or serialize itself into a text-based format. <p>Any attribute accessors that take or return a <code>String</code> value are performing the conversion for convenience.</p> <p>There are two ways to set the value of an attribute or simple value:</p> <ul style="list-style-type: none"> • Set the value using an <code>XmLSimpleValue</code> object using the <code>SimpleValue</code> property. • Set the value it in the natural Gosu simple type (such as <code>int</code> or <code>byte[]</code>) in the <code>Value</code> property. This only is an option if the element is an XSD-based type. (And for XSD types, you actually access this using the syntax <code>element.\$Value</code>) • Set the XML value in text-based format directly: <ul style="list-style-type: none"> • For attributes, call the <code>setAttributeValue</code> method. • For element simple values, set the <code>Text</code> property. For XSD types, you actually access this using the syntax <code>element.\$Text</code>. <p>For details, see “XSD-based Properties and Types” on page 281 and “Getting Data From an XML Element” on page 288.</p>

Feature	Legacy XML APIs	Current XML APIs
How does each element object treat attributes or content?	<p>The element contains associated attributes and content.</p>	<p>The element does not directly contain attributes and content. Elements have a backing type instance, which contains the attributes and content.</p> <p>In typical code, you can usually ignore this implementation detail because you can access type instance contents through properties and methods directly on <code>XmLElement</code>.</p> <p>Through this mechanism, each element exposes methods like <code>getChild</code> and <code>getChildren</code> and the <code>Children</code> property to get child elements.</p> <p>For XSD-based types, the Gosu type system adds various types and properties to each element. The properties on the element correspond to attributes (by name) and child elements (by name).</p> <p>If a child element is a simple type (or it is a complex type with simple content), Gosu adds two separate properties. Gosu adds one property for the value (get or set the simple type directly in Gosu). Gosu adds another property for the child element itself. The one for the element has an <code>_e1em</code> suffix. For example, a child element named <code>PublicID</code> would create a properties <code>PublicID</code> and <code>PublicID_e1em</code>.</p> <p>In contrast, if the child element is not a simple type (and not a complex type with simple content), Gosu creates the property with its natural name. In other words, with no <code>_e1em</code> suffix.</p> <p>For the full details, see "XSD-based Properties and Types" on page 281.</p>
Handling of comments in XML	<p>On parse (import), Gosu attempts to preserve comments. Access the comments on parsed XML in the <code>node.Comment</code> property. You can also set this property on the node. During serialization (export), Gosu serializes the comments. There might be differences from the original parsed XML. For example, multiple comments on the same element become one comment (containing text from both original comments).</p>	<p>On parse (import), Gosu does not import XML comments. To set a new comment on an element, set the <code>element.Comment</code> property. For XSD-based element types, set the <code>element.\$Comment</code> property instead.</p> <p>During XML element serialization (export), the comment, if any, appears before the element. There is an API to optionally suppress the comment. See "Exporting XML Data" on page 275.</p> <p>IMPORTANT: This difference in handling comments may be important for some use cases. For example, if you plan to process files that make great use of comments, including commenting-out sections to disable them temporarily. In these cases, you might consider using <code>XMLNode</code> rather than <code>XmLElement</code>.</p>

Feature	Legacy XML APIs	Current XML APIs
Handling of whitespace in XML	Whitespace is discarded.	For non-XSD elements, if an element contains children, then the whitespace is discarded during parse (import). Otherwise, it is retained and available through the <code>Text</code> property. For XSD-based elements, Gosu saves whitespace if and only if the element's XSD type is a simple type and the XSD specifies that parsers must preserve whitespace. This is controlled by the XSD whitespace facet. (See the XML schema specification for details.)
Naming convention for special APIs exposed as properties on a node/element	The regular name for the property.	If it is a non-XSD-based element type (for example using the APIs as untyped nodes), it is the regular name for the property. For XSD-based elements, the special APIs exposed on the type instance require you to add the dollar sign prefix (\$) to the property name. Instead of getting <code>element.Children</code> , you get <code>element.\$Children</code> . For details, see "Dollar Sign Prefix for Properties that Are XSD Types" on page 274.

See also

- “Legacy XML APIs: Exporting XML Data” on page 316
- “Legacy XML APIs: Manipulating XML as Untyped Nodes” on page 313
- “Legacy XML APIs: Collection-like Enhancements for XML” on page 318
- “Legacy XML APIs: Structured XML Using XSDs” on page 319
- “The Guidewire XML (GX) Modeler” on page 302

Introduction to the XML Element in Gosu

The main class that represents an XML element is the class `Xmlelement`.

An `Xmlelement` object consists of the following items (and only the following items):

- **The element name as a qualified name (QName)** – The element’s name is not simply a `String` value. It is a fully-qualified name called a `QName`. A `QName` represents a more advanced definition of a name than a simple `String` value. To define a `QName`, use the class `javax.xml.namespace.QName`. A `QName` object contains the following components:
 - A `String` value that represents the local part (also called the `localPart`)
 - A `String` value that represents the namespace URI that the local part of the name is defined within. For example, a namespace might have the value: `http://www.w3.org/2001/XMLSchema-instance`
 - A suggested prefix name if Gosu later serializes this element. (This prefix is not guaranteed upon serialization, since there may be conflicts.)

For example, you might see in an XML file an element name with the syntax of two parts separated by a colon, such as `veh:root`. The `root` part of the name is the local part. The prefix `veh` in this example indicates that the XML document (earlier in the file) declared a namespace and a shortcut name (the prefix `veh`) to represent the full URI.

For example, consider the following XML document:

```
<?xml version="1.0"?>
<veh:root xmlns:veh="http://mycompany.com/schema/vehiclexsd">
  <veh:childelement/>
</veh:root>
```

The following things are true about this XML document:

- The root element of the document has the name `root` within the namespace `http://mycompany.com/schema/vehiclexsd`.
- The text `xmlns:veh` followed by the URI means that later in the XML document, elements can use the namespace shortcut `veh`: to represent the longer URI: `http://mycompany.com/schema/vehiclexsd`.
- The root element has one child element, whose name is `childelement` within the namespace `http://mycompany.com/schema/vehiclexsd`. However, this XML document specifies the namespace not with the full URI but with the shortcut prefix `veh` followed by the colon (and then followed by the local part).

There are three constructors for `QName`:

- constructor specifying the namespace URI, local part, and suggested prefix.
`QName(String namespaceURI, String localPart, String prefix)`
- constructor specifying the namespace URI and local part (suggested prefix is implicitly empty).
`QName(String namespaceURI, String localPart)`
- constructor specifying the local part only (the namespace and URL are implicitly empty)
`QName(String localPart)`

You can set the namespace in the `QName` object to the empty namespace, which technically is the constant `javax.xml.XMLConstants.NULL_NS_URI`. The recommended approach for creating `QName` objects in the empty namespace is to use the `QName` constructor that does not take a namespace argument.

To create multiple `QName` objects easily in the same namespace, you can use the optional utility class called `XmNamespace`. For details, see “Creating Many Qualified Names in the Same Namespace” on page 280.

Whenever you construct an `XmElement`, the name is strictly required and must be non-empty.

Other Gosu XML APIs use `QName` objects for other purposes. For example, attributes on an element are names defined within a namespace, even if it is the default namespace for the XML document or the empty namespace. Gosu natively represents both attribute names and element names as `QName` objects.

- **A backing type instance** – Each element contains a reference to a Gosu type that represents this specific element. To get the backing type instance, get the `TypeInstance` property from the element. For XML elements that Gosu created based on an XSD, Gosu sets this backing type information automatically so it can be used in a typesafe manner.

Whenever you construct an `XmElement`, an explicit backing type is optional. If you are constructing the element from an XSD, Gosu sets the backing type automatically based on the subclass of `XmElement`.

You can use `XmElement` essentially as untyped nodes, in other words with no explicit XSD for your data format. If you are not using an XSD and do not provide a backing type, Gosu uses the default backing type `gw.xml.xsd.w3c.xmlschema.types.complex.AnyType`. All valid backing types are subclass of that `AnyType` type. See “Getting Data From an XML Element” on page 288 for related information

The type instance of an XML element is responsible for most of the element’s behavior but does not contain the element’s name. You can sometimes ignore the division of labor between an `XmElement` and its backing type instance. If you are using an XSD, this distinction is useful and sometimes critical. For more information, see “Getting Data From an XML Element” on page 288.

- **The nilness of the element** – XML has a concept of whether an element is `nil`. This is not exactly the same as being `null`. An element can be `nil` (and must have no child elements) but still have attributes. Additionally, an XSD can define whether an element is `nillable`, which means that element is allowed to be `nil`. For more information, see “Access the Nilness of an Element” on page 294.

To summarize, the `XmLElement` instance contains the properties shown in the following table.

<code>XmLElement</code> property	Type	Description
<code>QName</code>	<code>QName</code>	A read-only property that returns the element's in Gosu XML APIs.
<code>Namespace</code>	<code>XmLNamespace</code>	Returns an <code>XmLNamespace</code> object that represents the element's namespace
<code>TypeInstance</code>	<code>gw.xsd.w3c.xmlschema.types.complex.AnyType</code> or any subclass of that class	Returns the element's backing type instance
<code>Nilness</code>	<code>boolean</code>	Specifies whether this element is nil, which is an XML concept that is not the same as being null. See "Access the Nilness of an Element" on page 294.

IMPORTANT If you are accessing these properties on an XSD-based element, you must use a dollar sign prefix for the property name. See "Dollar Sign Prefix for Properties that Are XSD Types" on page 274

To create a basic `XmLElement`, simply pass the element name to the constructor as either a `QName` object or a `String`. The constructor on `XmLElement` that takes a `String` is a convenience method. The `String` constructor is equivalent passing a new `QName` object with that `String` as the one-argument constructor to `QName`. In other words, the namespace and prefix in the `QName` are `null` if you use the `String` constructor on `XmLElement`.

The following code creates an in-memory Gosu object that represents an XML element `<Root>` in the empty namespace:

```
var el = new XmLElement( "Root" )
```

In this case, the `el.TypeInstance` property returns an instance of the default type `gw.xsd.w3c.xmlschema.types.complex.AnyType`. If you instantiate a type instance, typically you would use more specific subclass of `AnyType`, either an XSD-based type or a simple type.

For a more complex example, the following Gosu code creates a new `XmLElement` without an XSD, and adds a child element:

```
uses gw.xml.XmLElement
uses javax.xml.namespace.QName

var e = new XmLElement(new QName("http://mycompany.com/schema/vehiclexsd", "root", "veh"))
var e2 = new XmLElement(new QName("http://mycompany.com/schema/vehiclexsd", "childelement", "veh"))

e.addChild(e2)
e.print()
```

This prints the following:

```
<?xml version="1.0"?>
<veh:root xmlns:veh="http://mycompany.com/schema/vehiclexsd">
  <veh:childelement/>
</veh:root>
```

This output is the `QName` example from earlier in this section.

For more information about adding child elements, see "Getting Data From an XML Element" on page 288.

What Does an Element Contain Inside It?

Gosu exposes properties and methods on the XML type instances to let you access or manipulate child elements or text contents. XML elements can contain two basic types of content:

- Child elements
- A simple value, which can represent simple types such as numbers or dates

Technically, the Gosu object that represents the element does not directly contain the child elements or the text content. It is the backing type instance for each element that contains the text content. However, in practice this distinction is not typically necessary to remember.

An element can contain either child elements or simple values, but never both at the same time. This distinction is important particularly for XSD-based types. Gosu handles properties on an element differently depending on whether the element contains a simple value or is a type that can contain child elements.

Dollar Sign Prefix for Properties that Are XSD Types

For some properties that the documentation mentions, Gosu provides access directly from the XML element even though the actual implementation internally is on the backing type instance. If an element is not an XSD-based element, simply access the properties directly, such as `element.Children`.

However, if you use an XSD type, you must prefix the property name with a dollar sign (\$). This convention prevents ambiguity with properties defined on the XSD type or on the type instance that backs that type. For example, suppose the XSD defines an element's child element as one named `Children`. There would unfortunately be two similar properties with the same name. Gosu prevents ambiguity by requiring the special properties to have a dollar sign prefix if and only if the element is XSD-based:

- To access the children of an XSD-based element, use the syntax `element.$Children`.
- To access the a custom child element named `Children` as defined by the XSD, use the syntax `element.Children`. This is a non-recommended name due to the ambiguity, but Gosu has no problem with it. You may not have control over the XSD format that you are using, so Gosu must disambiguate them.

Notes about this convention:

- This convention only applies to properties defined on XSD-based types.
- It does not apply to methods.
- It does not apply to non-XSD-based XML elements.

For example, suppose you use the root class `XmlElement` directly with no XSD to manipulate an untyped graph of XML nodes. In that case, you can omit the dollar sign because the property names are not ambiguous. There are no XSD types, so there is no overlap in namespace.

This affects the following type instance property names that appear on an XML element, listed with their dollar sign prefix:

- `$Attributes`
- `$Class`
- `$Children`
- `$Namespace`
- `$NamespaceContext`
- `$Comment`
- `$QName`
- `$Text`
- `$TypeInstance`
- `$SimpleValue`
- `$Value` – Only for elements with an XSD-defined simple content
- `$Nil` - only for XSD-defined nullable elements.

Note: If you create an `XmlElement` element directly, not a subclass, the object is not an XSD type. It is an untyped node that uses the default type instance, an instance of the type `AnyType`. In such cases, there is no dollar sign prefix because there is no ambiguity between properties that are really part of the type instance, rather than on the XSD type.

Also see “Access the Nullness of an Element” on page 294

Exporting XML Data

The `XmlElement` class includes the following methods and properties that export XML data. All of these methods have alternate method signatures that takes a serialization options object (`XmlSerializationOptions`). See later in this topic for details of this object.

Export-related Methods on an XML Element

Each XML element provides the following methods that serialize the XML element:

- `bytes` method – Returns an array of bytes (the type `byte[]`) that contains the UTF-8-encoded bytes in the XML. Generally speaking, the `bytes` method is the best approach for serializing the XML. For example:

```
var ba = element.bytes()
```

If your code sends XML with a transport that understands only character data and not byte data, always base-64 encode the bytes to compactly and safely encode binary data. For example:

```
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 payload field in a `Message` entity is type `String`.

- `print` method – Serializes the element to the standard output stream (`System.out`). For example:
`element.print()`
- `writeTo` method – Writes to an output stream (`java.io.OutputStream`) but does not close the stream afterward.
- `asUTFString` method – Serializes the element to a `String` object in UTF-8. For example:
`var s = element.asUTFString()`

The `asUTFString` method outputs the node as a `String` value that contains 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 detect the encoding (UTF-8 or UTF-16). The existence of a byte order mark at the beginning of the document tells the parser what encoding to use.

Although the `asUTFString` method is helpful for debugging use, the `asUTFString` method is not the best way to export XML safely to external systems. In general, use the `bytes` method to get an array of bytes. If your code sends or stores XML with a transport that only understands character data (not byte data), always Base64 encode the array of bytes. See the example earlier in this section for the `bytes` method.

For more details of the XML byte order mark, see <http://www.w3.org/TR/REC-xml/#sec-guessing>.

For more information about UTF-8, see <http://tools.ietf.org/html/rfc3629>.

For all serializations, test your code with non-English characters to assure your tests cover characters with high Unicode code points.

IMPORTANT Always test your XML serialization and integration code with non-English characters.

For all of these methods, you can customize serialization by optionally passing an `XmlSerializationOptions` instance as another parameter at the end of the parameter list.

XML Serialization Options Reference and Examples

The following table lists properties on a serialization options object of type `gw.xml.XmlSerializationOptions`.

Serialization options	Type	Description	Default
General properties			
Comments	Boolean	If true, exports each element's comments.	true
Sort	Boolean	If true, ensures that the order of children elements of each element match the XSD. This is particularly important for sequences. This feature only has an effect on an element if it is based on an XSD type. If the entire graph of <code>XmlElement</code> objects contains no XSD-based elements, this property has no effect. If a graph of XML objects contains a mix of XSD and non-XSD-based elements, this feature only applies to the XSD-based elements. This is true independent of whether the root node is an XSD-based element. WARNING: For large XML objects with many nested layers, sorting requires a lot of computer resources. See "Serialization Performance and Element Sorting" on page 277.	true
<code>XmlDeclaration</code>	Boolean	If true, writes the XML declaration at the top of the XML document.	true
Validate	Boolean	If true, validates the XML document against the associated XSD. This feature only has an effect on an element if it is based on an XSD type. If the entire graph of <code>XmlElement</code> objects contains no XSD-based elements, this property has no effect.	true
Encoding	Charset	The character encoding of the resulting XML data as a <code>java.nio.charset.Charset</code> object. See discussion after this table for tips for setting this property.	UTF-8 encoding
Pretty	Boolean	If true, Gosu attempts to improve visual layout of the XML with indenting and line separators. If you set this to false, then Gosu ignores the values of the properties: <code>Indent</code> , <code>LineSeparator</code> , <code>AttributeNewLine</code> , <code>AttributeIndent</code> .	true
Properties used only if Pretty property is true			
Indent	String	The String to export for each indent level to make the hierarchy clearer.	Two spaces
LineSeparator	String	The line separator as a String.	The new line character (ASCII 10).
AttributeNewLine	Boolean	If true, puts each attribute on a new line.	false
AttributeIndent	int	The number of additional indents beyond its original indent for an attribute.	2

In addition, `XmlSerializationOptions` exposes special methods for each one of these properties. Each method has the prefix `with` followed by the property name. For example, `withSort`. The method takes one argument of the type of that property as listed in the table.

These methods are chainable, which means that they return the `XmlSerializationOptions` object again. This means you can use code such as:

```
var opts = new gw.xml.XmlSerializationOptions().withSort(false).withValidate(false)
```

In addition, the `withEncoding` property has a secondary method signature that takes the Java short name for the encoding. This means there are two ways to set the encoding:

- Use the `withEncoding` method and pass a standard Java encoding name as a `String`, such as "Big5".

- Set the `Encoding` property to a raw character set object for the encoding. You can use the static method `Charset.forName(ENCODING_NAME)` to get the desired static instance of the character set object. For example, pass "Big5".

XML Serialization Examples

For example, the following example creates an element, then adds an element comment. Next, it demonstrates printing the element with the default settings (with comments) and how to customize the output to omit comments.

```
uses gw.xml.XmlSerializationOptions

// Create an element.
var a = new com.guidewire.pl.docexamples.gosu.xml.simpleelement.MyElement()
.

// Add a comment
a.$Comment = "Hello I am a comment"

print("print element with default settings...")
a.print()

print("print element with no comments...")
a.print(new XmlSerializationOptions() { :Comments = false } )
```

For Guidewire application messaging, follow the pattern of the following ClaimCenter Event Fired rule code. It creates a new message that contains the XML for a contact entity as a `String` to work with the standard message payload property. The messaging system requires a `String`, not an array of bytes. To properly and safely encode XML into a `String`, use the syntax:

```
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:

```
override function send(message : Message, transformedPayload : String) : void {

    // Decode the Base64 encoded bytes stored in a String.
    var bytes = Base64Util.decode(message.Payload)

    // Send the byte array to a foreign system.
    ...

    message.reportAck();
}
```

All serialization APIs generate XML data for the entire XML hierarchy with that element at the root. If you are using a GX model in Studio, optionally you can choose other export options, such as:

- **Verbose** – Export elements even for `null` properties using the `Verbose` property in `GXOptions`.
- **Incremental** – Export only entity fields that changed using the `Incremental` property in `GXOptions`.

See also

- “Customizing GX Modeler Output (`GXOptions`)” on page 308
- “The Guidewire XML (GX) Modeler” on page 302

Serialization Performance and Element Sorting

The element sorting serialization feature (`XmlSerializationOptions.Sort`) ensures that the order of children elements of each element match the XSD. This is particularly important for sequences.

For large XML objects with many nested layers, sorting requires a lot of computer resources.

If you create your XML objects in the order specified by their sequence definitions, then you can safely turn off sorting during serialization. For example.

```
myXMLElement.print(new XmlSerializationOptions() { :Sort = false })
```

Parsing XML Data into an XML Element

The `XmLElement` class contains static methods for parsing XML data into a graph of `XmLElement` objects. Parsing means to convert serialized XML data into a more complex in-memory representation of the document. All these methods begin with the prefix `parse`. There are multiple methods because Gosu supports parsing from several different sources of XML data.

IMPORTANT For each source of data, there is an optional method variant that modifies the way Gosu parses the XML. Gosu encapsulates these options in an instance of the type `XmlParseOptions`. The `XmlParseOptions` specifies additional schemas that resolve schema components for the input instance XML document. Typical code does not need this. Use this if your XML data contains references to schema components that are neither directly nor indirectly imported by the schema of the context type. For more information, see later in this topic.

For example, the following simple example parses XML contained in a `String` into an `XmLElement` object, and then prints the parsed XML data:

```
var a = XmLElement.parse("<Test123/>")
a.print()
```

If you are using an XSD, call the `parse` method directly on your XSD-based node, which is a subclass of `XmLElement`. For example:

```
var a = com.guidewire.p1.docexamples.gosu.xml.demoattributes.Element1.parse(xmlDataString)
```

The following table lists the parsing methods.

Method name	arguments	Description
parse	<code>byte[]</code> <code>byte[], XmlParseOptions</code>	Parse XML from a byte array with optional parsing options.
parse	<code>java.io.File</code> <code>java.io.File, XmlParseOptions</code>	Parse XML from a file, with optional parsing options.
parse	<code>java.io.InputStream</code> <code>java.io.InputStream, XmlParseOptions</code>	Parse XML from an <code>InputStream</code> with optional parsing options.

Method name	arguments	Description
parse	java.io.Reader java.io.Reader, XmlParseOptions	<p>Parse XML from a reader, which is an object for reading character streams. Optionally, add parsing options.</p> <p>WARNING: Because this uses character data, not bytes, the character encoding is irrelevant. Any encoding header at the top of the file has no effect. It is strongly recommended to treat XML as binary data, not as String data. If your code needs to send XML with a transport that only understands character (not byte) data, always Base64 encode the bytes. (From Gosu, use the syntax: <code>Base64Util.encode(element.bytes())</code>)</p>
parse	String String, XmlParseOptions	<p>Parse XML from a String, with optional parsing options.</p> <p>IMPORTANT: Because this uses character data, not bytes, the character encoding is irrelevant. Any encoding header at the top of the file has no effect. It is strongly recommended to treat XML as binary data, not as String data. If your code needs to send XML with a transport that only understands character (not byte) data, always Base64 encode the bytes. From Gosu, create the syntax: <code>Base64Util.encode(element.bytes())</code></p>

For details of `XmlParseOptions`, see “Referencing Additional Schemas During Parsing” on page 279.

Checking XML Well-Formedness and Validation During Parsing

For XSD-based XML elements, Gosu has the following behavior:

- Gosu checks for well-formedness (for example, no unclosed tags or other structural errors).
- Always validates the XML against the XSD.

For non-XSD-based XML elements:

- Gosu checks for well-formedness.
- If the XML parse options object includes references to other schemas, Gosu validates against those schemas.

If the XML document fails any of these tests, Gosu throws an exception.

See also

- “Referencing Additional Schemas During Parsing” on page 279.

Referencing Additional Schemas During Parsing

In some advanced parsing situations, you might need to reference additional schemas other than your main schema during parsing.

To specify additional schemas, set the `XmlParseOptions.AdditionalSchemas` to a specific `SchemaAccess` object. This `SchemaAccess` object represents the XSD. To access it from an XSD, use the syntax:

```
package_for_the_schema.util.SchemaAccess
```

To see how and why you would use this, suppose you have the following two schemas:

The XSD ImportXSD1.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="urn:ImportXSD1"
    xmlns:ImportXSD1="urn:ImportXSD1">
    <xsd:element name="ElementFromSchema1" type="ImportXSD1:TypeFromSchema1"/>
    <xsd:complexType name="TypeFromSchema1"/>
</xsd:schema>
```

The XSD ImportXSD2.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="urn:ImportXSD2"
    xmlns:ImportXSD1="urn:ImportXSD1" xmlns:ImportXSD2="urn:ImportXSD2"
    elementFormDefault="qualified">
    <xsd:import schemaLocation="ImportXSD1.xsd" namespace="urn:ImportXSD1"/>
    <xsd:complexType name="TypeFromSchema2">
        <xsd:complexContent> <!-- the TypeFromSchema2 type extends the TypeFromSchema1 type! -->
            <xsd:extension base="ImportXSD1:TypeFromSchema1"/>
        </xsd:complexContent>
    </xsd:complexType>
</xsd:schema>
```

Notice that the ImportXSD2 XSD extends a type that the ImportXSD1 defines. This is analogous to saying the ImportXSD2 type called TypeFromSchema2 is like a subclass of the ImportXSD1 type called TypeFromSchema1.

The following code fails (throws exceptions) because the ImportXSD1 references the schema type ImportXSD2:TypeFromSchema2 and Gosu cannot find it anywhere in the current schema.

```
var schema2 = com.guidewire.pl.docexamples.gosu.xml.importxsd2.util.SchemaAccess

var xsdtext = "<ElementFromSchema1 xmlns=\"urn:ImportXSD1\" xmlns:ImportXSD2=\"urn:ImportXSD2\""" +
    " xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" xsi:type=\"ImportXSD2:TypeFromSchema2\"/>

// Parse an element defined in the the first schema, but pass an extension to that
// type that the second schema defines. THIS FAILS without using the AdditionalSchemas feature.
var element = ElementFromSchema1.parse(xsdtext)
```

The main problem is that the ImportXSD1 XSD type does not directly know about the existence of the schema called ImportXSD2 even though it extends one of its types.

To make it work, set the AdditionalSchemas property of the XmlParseOptions object to a list containing one or more SchemaAccess objects. In other words, the following XML parsing code succeeds:

```
var schema2 = com.guidewire.pl.docexamples.gosu.xml.importxsd2.util.SchemaAccess
var options = new gw.xml.XmlParseOptions() { :AdditionalSchemas = { schema2 } }

var xsdtext = "<ElementFromSchema1 xmlns=\"urn:ImportXSD1\" xmlns:ImportXSD2=\"urn:ImportXSD2\""" +
    " xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" xsi:type=\"ImportXSD2:TypeFromSchema2\"/>

// Parse an element defined in the the first schema, but pass an extension to that
// type that the second schema defines by using the XmlParseOptions.
var element = ElementFromSchema1.parse(xsdtext, options)
```

To remap an external namespace or XSD URL to a local XSD using the schemalocations.xml, see “Use a Local XSD for an External Namespace or XSD Location” on page 301.

Creating Many Qualified Names in the Same Namespace

The name of each element is a qualified name, which is an object of type javax.xml.namespace.QName.

A QName object contains the following parts:

- A namespace URI
- A local part
- A suggested prefix for this namespace.

On serialization of an XElement, Gosu tries to use the prefix to generate the name, such as “wsdl:definitions”. In some cases however, it might not be possible to use this name. For example, if an XML element defines two attributes with different namespaces but the same prefix. On serialization, Gosu auto-creates a prefix for one of them to prevent conflicts.

Typical code repetitively creates many QName objects in the same namespace. One way to create many such objects is to store the namespace URI in a String variable, then create QName instances with new local parts.

To simplify this process, Gosu includes a utility class called `gw.xml.XmlNamespace`. It represents a namespace URI and a suggested prefix. In other words, it is like a QName but without the local part.

There are two ways to use `XmlNamespace`:

- Create an `XmlNamespace` directly and call its `qualify` method and pass the local part String. For example:

```
uses gw.xml.XmlNamespace
var ns = new XmlNamespace("namespaceURI", "prefix")
var e = new XmlElement(ns.qualify("localPartName"))

// Create a new XML element.
var xml = new XmlElement(new QName("namespaceURI", "localPart", "prefix"))

// Reuse the namespaceURI and prefix from the previously-created element.
var xml2 = new XmlElement(xml.Namespace.qualify("localPart2"))
```

For more information, see “Introduction to the XML Element in Gosu” on page 271.

XSD-based Properties and Types

The most powerful way to use XML in Gosu is to use an XSD that describes in a strict way what is valid in your XML. If you can use or generate an XSD for your data, it is strongly recommended to use an XSD.

To tell Gosu to load your XSD, put your XSD files in the same file hierarchy in the `configuration` module as Gosu classes, organized in subdirectories by package. At the file system level, this is the hierarchy of files starting at:

```
ClaimCenter/modules/configuration/gsrc
```

For example, to load an XSD in the package `mycompany.schemas`, put your XSD file at the path:

```
ClaimCenter/modules/configuration/gsrc/mycompany/schemas
```

Gosu creates new types in the type system for element declarations in the XSD. Where appropriate, Gosu creates properties on these types based on the names and structure within the XSD. By using an XSD and the generated types and properties, your XML-related code is significantly easier to read and understand. For example, you can use natural Gosu syntax to access child elements by their name such as `element.ChildName` for a child named `ChildName`.

If you cannot use an XSD, you can use the basic properties and methods of `XmlElement` like `element.Children` and `element.getChild("ChildName")`. However, writing XML-related code without XSD types tends to be harder to understand getting and setting values and elements, and much less typesafe.

Important Concepts in XSD Properties and Types

There are some important distinctions to make in terminology when understanding how Gosu creates types from XSDs. In the following table, note how every definition in the XSD has a corresponding instance in an XML document (although in some cases might be optional).

Definitions (in the XSD)	Instances (in an XML document)
a schema (an XSD)	XML document
<code>element</code> definition	<code>element</code> instance
<code>complex type</code> definition	<code>complex type</code> instance
<code>simple type</code> definition	<code>simple type</code> instance
<code>attribute</code> definition	<code>attribute</code> instance

For every element definition in the XSD:

- There is an associated type definition.
- The type definition is either a complex type definition or simple type definition.
- The element definition has one of the following qualities:
 - It *references* a top-level type definition (for example, a top-level complex type)
 - It *embeds* a type definition inside the element definition (for example, an embedded simple type)
 - It includes no type, which implicitly refers to the built-in complex type `<xsd:anyType>`

In an XSD, various definitions cause Gosu to create new types:

- An element definition causes Gosu to create a type that describes the element
- A type definition causes Gosu to create a type that describes the type (for example, a new complex type)
- An attribute definition causes Gosu to create a type that describes the attribute

For example, suppose an XSD declares a new top-level simple type that represents a phone number. Suppose there are three element definitions that reference this new simple type in different contexts for phone numbers, such as work number, home numbers, and cell number. In this example, Gosu creates:

- One type that represents the phone number simple type
- Three types that represent the individual element definitions that reference the phone number

From Gosu, whenever you create objects or set properties on elements, it is important to know which type you want to use. In some cases, you might be able to do what you want in more than one way, although one way might be easier to read. See “XSD Generated Type Examples” on page 285 for examples that illustrate this point further.

Also remember that if you have a reference to the element, you can always reference the backing type. For example, for an element, you can reference the backing type instance using the `$TypeInstance` property. See “XSD Generated Type Examples” on page 285 for examples of this.

Reference of XSD Properties and Types

The following table lists the types and properties that Gosu creates from an XSD. For this topic, `schema` represents the fully-qualified path to the schema, `elementName` represents an element name, and `parentName` and `childName` represent names of parent and child elements.

The rightmost column indicates (for properties only) whether the property becomes a list property if it can appear more than once. If it says “Yes”, the property has type `java.util.List` parameterized on what type it is when it is singular. For example, suppose a child element is declared in the XSD with the type `xsd:int`:

- If its `maxOccurs` is 1, the property’s type is `Integer`.
- If its `maxOccurs` is greater than 1, the property’s type is `List<Integer>`, which means a list of integers.

There are other circumstances in which a property becomes a list. For example, suppose there is a XSD choice (`<xsd:choice>`) in an XSD that has `maxOccurs` attribute value greater than 1. Any child elements become list properties. For example, if the choice defines child elements with names “`elementA`” and “`elementB`”, Gosu creates properties called `ElementA` and `ElementB`, both declared as lists. Be aware that Gosu exposes shortcuts for inserting items, see “Automatic Insertion into Lists” on page 286.

Notes about generated types containing the text `anonymous` in the fully qualified type name:

- Although the package includes the word `anonymous`, this does not imply that these elements have no defined names. The important quality that distinguishes these types is that the object is defined at a lower level than the top level of the schema. By analogy, this is similar to how Gosu and Java define inner classes within the namespace of another class.
- There are several rows that contain a reference to the path from root as the placeholder text `PathFromRoot`. The path from root is a generated name that embeds the path from the root of the XSD, with names separated

by underscore characters. The intermediate layers may be element names or group names. See each row for examples.

For each occurrence of...	Declared in the schema at this location...	There is a new...	With syntax...
element definition	top level	type	<code>schema.ElementName</code> IMPORTANT: However, Gosu behaves slightly differently if the top-level element is declared in a web service definition language (WSDL) document. Instead, Gosu creates the type name as <code>schema.elements.ElementName</code> .
	lower than top level	type	<code>schema.anonymous.elements.PathFromRoot_ElementName</code> For example, suppose the top level group A that contains an element called B, which contains an element called C. The <code>PathFromRoot</code> is <code>A_B</code> and the fully-qualified type is <code>schema.anonymous.elements.A_B_C</code> .
complex type definition	top level	type	<code>schema.types.complex.TypeName</code>
	lower than top level	type	<code>schema.anonymous.types.complex.PathFromRoot</code> For example, suppose a top level element A contains an embedded complex type. The <code>PathFromRoot</code> is A. Note that complex types defined at a level lower than the top level never have names.
simple type definition	top level	type	<code>schema.types.simple.TypeName</code>
	lower than top level	type	<code>schema.anonymous.types.simple.PathFromRoot</code> For example, suppose a top level element A contains element B, which contains an embedded simple type. The path from root is <code>A_B</code> . Note that simple types defined at a level lower than the top level never have names.
attribute definition	top level	type	<code>schema.attributes.AttributeName</code>
	lower than top level	type	<code>schema.anonymous.attributes.PathFromRoot</code> For example, suppose a top level element A contains element B, which has the attribute C. The path <code>PathFromRoot</code> is <code>A_B</code> and the fully-qualified type is <code>schema.anonymous.attributes.A_B_C</code> .
	within an element	property	<code>element.AttributeName</code> Unlike most other generated properties on XSD types, an attribute property never transform into a list property.

For every child element with either (1) simple type or (2) complex type and a simple content

It is a common pattern to convert a `simpleType` at a later time to `simpleContent` simply to add attributes to an element with a simple type. To support this common pattern, Gosu creates two properties `ChildName` and `ChildName_e1em` for every child element with either a simple type or both a complex type and simple content. The one with the `_e1em` suffix contains the element object instance. The property without the `_e1em` suffix contains the element value. Because of this design, if you later decide to add attributes to a `simpleType` element, your XML code requires no changes simply because of this change.

child element with either:	anywhere	property	<code>element.ChildName_e1em</code> The property type is as follows: <ul style="list-style-type: none"> • If element is defined at top-level, <code>schema.ElementName</code> • If element is defined at lower levels, <code>schema.anonymous.elements.PathFromRoot_ElementName</code>.
			IMPORTANT: This property transforms into a list type if it can appear more than once. See discussion of list properties immediately before this table.

For each occurrence of...	Declared in the schema at this location...	There is a new...	With syntax...
the value of a child element with either: <ul style="list-style-type: none">• simple type• complex type and a simple content	anywhere	property	<p><code>element.ChildName</code></p> <p>The property type is as follows:</p> <ul style="list-style-type: none"> • If element is defined at top-level, <code>schema.ElementName</code> • If element is defined at lower levels, <code>schema.anonymous.elements.PathFromRoot_ElementName</code>. <p>IMPORTANT: This property transforms into a list type if it can appear more than once. See discussion of list properties immediately before this table.</p>
For every child element with complex type and no simple content			
child element with complex type and no simple content	anywhere	property	<p><code>element.ChildName</code></p> <p>The property type is as follows:</p> <ul style="list-style-type: none"> • If element is defined at top-level, <code>schema.ElementName</code> • If element is defined at lower levels, <code>schema.anonymous.elements.PathFromRoot_ElementName</code>. <p>IMPORTANT: This property transforms into a list type if it can appear more than once. See discussion of list properties immediately before this table.</p>
For each schema			
schema definition	n/a	schema access object	<p><code>schema.util.SchemaAccess</code></p> <p>It is a special utility object for providing access to the original schema that produced this type hierarchy. Think of this as a way of representing this schema. This is important if you need one schema to include another schema (see “Referencing Additional Schemas During Parsing” on page 279).</p>

Normalization of Gosu Generated XSD-based Names

In cases where Gosu creates type names and element names, Gosu performs slight normalization of the names:

- One prominent aspect of normalization is capitalization to conform to Gosu naming standards for packages, properties, and types. For example, Gosu packages become all lowercase. Types must start with initial capitals. Properties must start with initial capitals.
- If the type or property names contains invalid characters for Gosu for that context, Gosu changes them. For example, hyphens are disallowed and removed.
 - If Gosu finds an invalid character and the following character is lowercase, Gosu removes the invalid character and uppercases the following letter.
 - If Gosu finds an invalid character and the following character is uppercase, Gosu converts the invalid character to an underscore and does not change the following character.
 - If the first character is invalid as a first character but otherwise valid (for example, a numeric digit), Gosu simply prepends an underscore. If it is entirely invalid within a name in that context (such as hyphen), Gosu removes the character. In the unusual case in which after removing all start characters, no characters remain, Gosu simply renames that item a simple underscore.
- If there are duplicates, Gosu appends numbers to some of them. For example, MyProp, MyProp2, MyProp3, and so on.

XSD Generated Type Examples

XSD Generated Type Examples 1

Let us try these with actual examples. Suppose you have the following XSD in the package `examples.pl.gosu.xml`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Child1"/> <!-- The default type is xsd:anyType. -->
        <xsd:element name="Child2" type="xsd:int"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Review the following Gosu code:

```
var xml = new packagename.myschema.Element1()
var child1 = xml.Child1           // Child1 has type schema.anonymous.elements.Element1_Child1.
var child2 = xml.Child2           // Child2 has type java.lang.Integer.
xml.Child2 = 5                  // Set the XML property with a simple type.
var child2Elem = xml.Child2_elem // Get the XML property as a
                                // schema.anonymous.elements.Element1_Child2.
```

Note the following:

- The `Child1` property is of type `schema.anonymous.elements.Element1_Child1`, which is a subclass of `XmlElement`.
- The `Child2` property is of type `java.lang.Integer`. When a child element has a simple type, its natural property name gets the object's value, rather than the child element object. If you wish to access the element object (the `XmlElement` instance) for that child, instead use the property with the `_elem` suffix. In this case, for the child named `Child2`, you use the `element.Child2_elem` property, which is of type `schema.anonymous.elements.Element1_Child2`.

XSD Generated Types: Element Type Instances Compared to Backing Type Instances

Suppose you have a XSD that defines one phone number simple type and multiple elements that use that simple type.

The XSD might look like the following:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="person">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="cell" type="phone"/>
        <xsd:element name="work" type="phone"/>
        <xsd:element name="home" type="phone"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="phone">
    <xsd:sequence>
      <xsd:element name="areaCode" type="xsd:string"/>
      <xsd:element name="mainNumber" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Suppose you want to create and assign the phone numbers. There are multiple ways to do this.

If you want to create three different phone numbers, use code like this:

```
var e = new schema.Person()

e.Cell.AreaCode = "415"
e.Cell.MainNumber = "555-1213"

e.Work.AreaCode = "416"
e.Work.MainNumber = "555-1214"
```

```
e.Home.AreaCode = "417"
e.Home.MainNumber = "555-1215"
```

In contrast, you want to create one phone number to use in multiple elements, you might use code like this:

```
var e = new schema.Person()

var p = new schema.types.complex.Phone()
p.AreaCode = "415"
p.MainNumber = "555-1212"

e.Cell.$TypeInstance = p
e.Work.$TypeInstance = p
e.Home.$TypeInstance = p
```

An element's `$TypeInstance` property accesses the element's backing type instance.

It is important to note that it is necessary to use the `$TypeInstance` property syntax because the Gosu declared types of each phone number element are incompatible.

For example, you cannot create the complex type and directly assign it to the element type:

```
var e = new schema.Person()

var p = new schema.types.complex.Phone()
p.AreaCode = "415"
p.MainNumber = "555-1212"

e.Cell = p // SYNTAX ERROR: cannot assign complex type instance to element type instance
e.Work = p // SYNTAX ERROR: cannot assign complex type instance to element type instance
e.Home = p // SYNTAX ERROR: cannot assign complex type instance to element type instance
```

Additionally, different element-based types can be mutually incompatible for assignment even if they are associated with the XSD type definition. For example:

```
var e = new schema.Person()

e.Cell = e.Work // SYNTAX ERROR: cannot assign one element type to a different element type
```

Automatic Insertion into Lists

If you are using XSDs, for properties that represent child elements that can appear more than once, Gosu exposes that property as a list. For properties that Gosu exposes as list properties (see “XSD-based Properties and Types” on page 281), Gosu has a special shorthand syntax for inserting items into the list. If you assign to the list index equal to the size of the list, then the index assignment becomes an insertion.

This is also true if the size of the list is zero: use the `[0]` array/list index notation and set the property. This inserts the value into the list, which is equivalent to adding an element to the list. However, you do not have to worry about whether the list exists yet if you use this syntax. (If you are creating XML objects in Gosu, by default the lists do not yet exist. From Gosu they are `null`.)

In other words, you can add an element with the syntax:

```
element.PropertyName[0] = childElement
```

If the list does not exist yet for a list property at all, Gosu creates the list upon the first insertion.

In other words, suppose an element contains child elements that represent an address and the child element has the name `Address`. If the XSD declares the element could exist more than once, the `element.Address` property is a list of addresses. The following code creates a new `Address`:

```
element.Address[0] = new my.package.xsdname.elements.Address()
```

Note: If you use XSDs, Gosu automatically creates intermediate XML elements as needed. Use this feature to significantly improve the readability of your XML-related Gosu code.

Example XSD

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Child1" type="xsd:int" maxOccurs="unbounded"/>
```

```
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Code

```
var xml = new schema.Element1()
print("Before insertion: ${xml.Child1.Count}")
xml.Child1[0] = 0
xml.Child1[1] = 1
xml.Child1[2] = 2
print("After insertion: ${xml.Child1.Count}")
xml.print()
```

Output

```
Before insertion: 0
After insertion: 3
<?xml version="1.0"?>
<Element1>
  <Child1>0</Child1>
  <Child1>1</Child1>
  <Child1>2</Child1>
</Element1>
```

Example XSD

This also works with simple types derived by list (xsd:list):

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Child1">
          <xsd:simpleType>
            <xsd:list itemType="xsd:int"/>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Output

```
Before insertion: 0
After insertion: 3
<?xml version="1.0"?>
<Element1>
  <Child1>0 1 2</Child1>
</Element1>
```

XSD List Property Example

If the possibility exists for a child element name to appear multiple times, then the property becomes a list-based property.

Example XSD

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="Child1" type="xsd:int"/>
        <xsd:sequence maxOccurs="unbounded">
          <xsd:element name="Child2" type="xsd:int"/>
        </xsd:sequence>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Code

```
var xml = new schema.Element1()
xml.Child1 = 1
```

```

xml.print()
print( "-----" )
xml.Child1 = null
xml.Child2 = {1, 2, 3, 4}
xml.print()

```

Output

```

<?xml version="1.0"?>
<Element1>
  <Child1>1</Child1>
</Element1>
-----
<?xml version="1.0"?>
<Element1>
  <Child2>1</Child2>
  <Child2>2</Child2>
  <Child2>3</Child2>
  <Child2>4</Child2>
</Element1>

```

Getting Data From an XML Element

The main work of an XML element happens in the type instance associated with each XML element. The type instance of an XML element is responsible for nearly all of the element behavior but does not contain the element's name. You can usually ignore the division of labor between an `XmlElement` and its backing type instance. If you are using an XSD, this distinction is useful.

If you instantiate a type instance, typically you use more specific subclass of `gw.xsd.w3c.xmlschema.types.complex.AnyType`.

Gosu exposes properties and methods on the XML type instances for you to get child elements or simple value.

It is important to note that XML elements contain two basic types of content:

- Child elements
- Simple values

An element can contain either child elements or a simple value, but not both at the same time.

Also see “Introduction to the XML Element in Gosu” on page 271.

Manipulating Elements and Values (Works With or Without XSD)

To get the child elements of an element, get its `Children` property. The `Children` property contains a list (`java.util.List<XmlElement>`) of elements. If this XML element is an XSD-based type, you must add the property name prefix `$`, so instead get the property called `$Children`.

If the element has no child elements, there are two different cases:

- If an element has no child elements and no text content, the `Children` property contains an empty list.
- If an element has no child elements but has text content, the `Children` property contains `null`.

To add a child element, call the parent element's `addChild` method and pass the child element as a parameter.

For example, suppose you had the following XSD:

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Child1"/> <!-- default type is xsd:anyType -->
        <xsd:element name="Child2" type="xsd:int"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

Things to notice in this XSD:

- The element named `Child1` has no explicit type. This means the default type applies, which is `xsd:anyType`.
- The element named `Child2` has the type `xsd:int`. This means that by definition, this element must contain an integer value. Integer is a simple type. Without the integer value (if it were empty or `null`), any XML for this document would be invalid according to the XSD.

If you have a reference to an XML element of a simple type, you can set its value by setting its `SimpleValue` property. (If you are using an XSD, add the dollar sign prefix: `$SimpleValue`)

To set a simple value, like an integer value for an element, there are several approaches:

- Set the value in the `SimpleValue` property, to a subclass of `XmlSimpleValue`. This allows you to directly create the simple value that Gosu stores in the pre-serialized graph of XML elements. If it is on an XSD type, specify the property name with the dollar sign prefix: `$SimpleValue`. To create an instance of the `XmlSimpleValue` of the appropriate type, call static methods on the `XmlSimpleValue` type with method names that start with `make....` For example, call the `makeIntInstance` method and pass it an `Integer`. It returns an `XmlSimpleValue` instance that represents an integer, and internally contains an integer. In memory, Gosu stores this information as a non-serialized value. Only during serialization of the XML, such as exporting into a byte array or using the debugging `print` method, does Gosu serialize the `XmlSimpleValue` into bytes or encoded text. For a full reference of all the simple value methods and all their variants, see “Simple Values” on page 291.
- To create simple text content (text simple value), set the element’s `Text` property to a `String` value. If it is on an XSD type, specify the property name with the dollar sign prefix: `$Text`.
- If you are using an XSD, you can set the natural value in the `Value` property. If it is on an XSD type, specify the property name with the dollar sign prefix: `$Value`. For example, use natural-looking code like `e.$Value = 5`. If you are using an XSD and have non-text content, this approach tends to result in more natural-looking Gosu code than creating instances of `XmlSimpleValue`.
- If you are using an XSD, Gosu provides a simple syntax to get and set child values with simple types. For example, set numbers and dates from an element’s parent element using natural syntax using the child element name as a property accessor. This lets you easily access the child element’s simple value with very readable code. For example, `e.AutoCost = 5`. See “XSD-based Properties and Types” on page 281.

The following Gosu code adds two child elements, sets the value of an element using the `Value` property and the `SimpleValue` property, and then prints the results. In this example, we use XSD types, so we must specify the special property names with the dollar sign prefix: `$Value` and `$SimpleValue`.

```
uses gw.xml.XmlSimpleValue

// Create a new element, whose type is in the namespace of the XSD.
var e = new com.guidewire.pl.docexamples.gosu.xml.demochildprops.Element()
var c = e.$Children // returns an empty list of type List<XmlElement>
print("Children " + c.Count + c)
print("")

// Create a new CHILD element that is legal in the XSD, and add it as child.
var c1 = new com.guidewire.pl.docexamples.gosu.xml.demochildprops.anonymous.elements.Element1_Child1()
e.addChild(c1)

// Create a new CHILD element that is legal in the XSD, and add it as child.
var c2 = new com.guidewire.pl.docexamples.gosu.xml.demochildprops.anonymous.elements.Element1_Child2()
print("before set: " + c2.$Value) // prints "null" -- it is uninitialized

c2.$SimpleValue = XmlSimpleValue.makeIntInstance(5)
print("after set with $SimpleValue: " + c2.$Value)

c2.$Value = 7
print("after set with $Value: " + c2.$Value)
print("")

// Add the child element.
e.addChild(c2)

c = e.$Children // Return a list of two child elements
print("Children " + c.Count + c)
```

```
print("")  
e.print()
```

This code prints the following:

```
Children 0[]  
  
before set: null  
after set with $SimpleValue: 5  
after set with $Value: 7  
  
Children 2[com.guidewire.pl.docexamples.gosu.xml.demochildprops.anonymous.elements.Element1_Child1  
instance, com.guidewire.pl.docexamples.gosu.xml.demochildprops.anonymous.elements.Element1_Child2  
instance]  
  
<?xml version="1.0"?>  
<Element1>  
  <Child1/>  
  <Child2>7</Child2>  
</Element1>
```

Note that the `Child2` element contains the integer as text data in the serialized XML export. Gosu does not serialize the simple types to bytes (or a `String`) until serialization. In this example, the final `print` statement is what serializes the element and all its subelements.

Getting Child Elements By Name

If you want to iterate across the `List` of child elements to find your desired data, you can do so using the `Children` property mentioned earlier in this topic. Depending on what you are doing, you might want to use the Gosu enhancements on lists to find the items you want. See “Collections” on page 251 for more details.

However, it is common to want to get a child element by its name. To support this common case, Gosu provides methods on the XML element object. There are two main variants of this method. Use `getChild` if you expect only one match. Use `getChildren` if you expect multiple matches. Each one of these has an alternate signature that takes a `String`.

- `getChild(QName)` – Searches the content list for a single child with the specified `QName` name. There is an alternate method signature that takes a `String` value for the local part name. For that method signature, Gosu internally creates a `QName` with an empty namespace and the specified local part name. This method requires there to be exactly one child with this name. If there are multiple matches, the method throws an exception. If there might be multiple matches, use the `getChildren` method instead.
- `getChildren(QName) : List` – Searches the content list for all children with the specified `QName` name. There is an alternate method signature that takes a `String` value for the local part name. For that method signature, Gosu internally creates a `QName` with an empty namespace and the specified local part name.

Reusing the code from the previous example, you could add the following lines to get the second child element by its name:

```
// Get a child using the empty namespce by passing a String.  
var getChild1 = e.getChild("Child1")  
  
// Get a child using a QName, and "reuse" the namespace of a previous node.  
var getChild2FromQName = e.getChild(getChild1.Namespace.qualify("Child2"))  
  
print(getChild2FromQName.asUTFString())
```

Output

```
<?xml version="1.0"?>  
<Child2>5</Child2>
```

Removing Child Elements By Name

To remove child elements, Gosu provides methods on the XML element to remove a child and specifying the child to remove by its name. Use `removeChild` if you expect only one match. Use `removeChildren` if you expect multiple matches.:

- `removeChild(QName)` : `XmLElement` – Removes the child with the specified QName name. There is an alternate method signature that takes a `String` value for the local part name. For that method, Gosu internally creates a QName with an empty namespace and the specified local part name.
- `removeChildren(QName)` : `List<XmLElement>` – Removes the child with the specified QName name. There is an alternate method signature that takes a `String` value for the local part name. For that method, Gosu internally creates a QName with an empty namespace and the specified local part name.

Attributes

Attributes are additional metadata on an element. For example, in the following example an element has the `color` and `size` attributes:

```
<myelement color="blue" size="huge">
```

Every type instance contains its attributes, which are `XmLSimpleValue` instances specified by a name (a QName).

Each `XmLElement` object contains the following methods and properties related to attributes of the element:

- `AttributeNames` property – Gets a set of QName objects. The property type is `java.util.Set<QName>`.
- `getAttributeSimpleValue(QName)` – Get attribute simple value by its name, specified as a QName. Returns a `XmLSimpleValue` object. There is an alternate method signature that takes a `String` instead of a QName, and it assumes an empty namespace.
- `getAttributeValue(QName) : String` – Get attribute value by its name, specified as a QName. Returns a `String` object. There is an alternate method signature that takes a `String` instead of a QName, and it assumes an empty namespace.
- `setAttributeSimpleValue(QName, XmLSimpleValue)` – Set attribute simple value by its name (as a QName) and its value (as a `XmLSimpleValue` object). There is an alternate method signature that takes a `String` instead of a QName, and it assumes an empty namespace.
- `setAttributeValue(QName, String)` – Set attribute value by its name (as a QName) and its value (as a `XmLSimpleValue` object). There is an alternate method signature that takes a `String` instead of a QName, and it assumes an empty namespace.

Using the previous example, the following code gets and sets the attributes:

```
myelement.setAttributeValue("color", XmLSimpleValue.makeStringInstance("blue"))
var s = myelement.getAttributeValue("size")
```

Generally speaking, if you use XSDs for your elements, for typical use do not use these APIs. Instead, use the shortcuts that Gosu adds. They provide a natural and concise syntax for getting and setting attributes.

See also

- “Simple Values” on page 291
- “XSD-based Properties and Types” on page 281

Simple Values

Gosu represents the XML format simple values with the `gw.xml.XmLSimpleValue` type. An `XmLSimpleValue` is a Gosu object that encapsulates a value and the logic to serialize that value to XML. However, until serialization occurs, Gosu may internally store it in a format other than `java.lang.String`.

For example, XML represents hexadecimal-encoded binary data using the XSD type `xsd:hexBinary`. Gosu represents an `xsd:hexBinary` value with an `XmLSimpleValue` whose backing storage is an array of bytes (`byte[]`), one byte for each byte of binary data. Only at the time any Gosu code serializes the XML element does Gosu convert the byte array to hexadecimal digits.

The following properties are provided by `XmLSimpleValue`:

- `GosuValueType` – The `IType` of the Gosu value

- **GosuValue** – The type-specific Gosu value, for example, a `javax.xml.namespace.QName` for an `xsd:QName`
- **StringValue** – A string representation of the simple value, which may not be the string that is actually serialized, such as with a `QName`

For more information about the role of simple values in Gosu XML APIs, see “Getting Data From an XML Element” on page 288.

Methods to Create XML Simple Values

The following table lists static methods on the `XmlSimpleValue` type that create `XmlSimpleValue` instances of various types.

Method signature	Description
<code>makeStringInstance(java.lang.String)</code>	Make String instance
<code>makeAnyURIInstance(java.net.URI)</code>	Make URI instance
<code>makeBooleanInstance(java.lang.Boolean)</code>	Make boolean instance
<code>makeByteInstance(java.lang.Byte)</code>	Make byte instance
<code>makeUnsignedByteInstance(java.lang.Short)</code>	Make unsigned byte instance
<code>makeDateInstance(gw.xml.date.XmlDate)</code>	Make date-time instance from an <code>XmlDate</code>
<code>makeDateTimeInstance(gw.xml.date.XmlDateTime)</code>	Make date instance from an <code>XmlDateTime</code>
<code>makeDecimalInstance(java.math.BigDecimal)</code>	make decimal instance from a <code>BigDecimal</code>
<code>makeDoubleInstance(java.lang.Double)</code>	make decimal instance from a <code>Double</code>
<code>makeDurationInstance(gw.xml.date.XmlDuration)</code>	Make duration instance
<code>makeFloatInstance(java.lang.Float)</code>	Make float instance
<code>makeGDayInstance(gw.xml.date.XmlDay)</code>	Make GDay instance
<code>makeGMonthDayInstance(gw.xml.date.XmlMonthDay)</code>	Make GMonthDay duration instance
<code>makeGMonthInstance(gw.xml.date.XmlMonth)</code>	Make GMonth instance
<code>makeGYearInstance(gw.xml.date.XmlYear)</code>	Make GYear instance
<code>makeGYearMonthInstance(gw.xml.date.XmlYearMonth)</code>	Make GYearMonth instance
<code>makeHexBinaryInstance(byte[])</code>	Make hex binary instance from byte array
<code>makeIDInstance(java.lang.String)</code>	Make IDInstance instance from a String
<code>makeIDREFInstance(gw.xml.XmlElement)</code>	Make IDREF instance
<code>makeIntegerInstance(java.math.BigInteger)</code>	Make big integer instance
<code>makeIntInstance(java.lang.Integer)</code>	Make integer instance
<code>makeLongInstance(java.lang.Long)</code>	Make long integer instance
<code>makeUnsignedIntInstance(java.lang.Long)</code>	Make unsigned integer instance
<code>makeUnsignedLongInstance(java.math.BigInteger)</code>	Make unsigned long integer instance
<code>make QNameInstance(javax.xml.namespace.QName)</code>	Make QName instance
<code>make QNameInstance(java.lang.String, javax.xml.namespace.NamespaceContext)</code>	Make QName instance from a standard Java namespace context. A namespace context object encapsulates a mapping of XML namespace prefixes and their definitions (namespace URIs). You can get an instance of <code>NamespaceContext</code> from an <code>XmlElement</code> its <code>NamespaceContext</code> property. The <code>String</code> argument is the qualified local name (including the prefix) for the new <code>QName</code> .
<code>makeShortInstance(java.lang.Short)</code>	Make duration instance
<code>makeUnsignedShortInstance(java.lang.Integer)</code>	Make unsigned short integer instance
<code>makeTimeInstance(gw.xml.date.XmlTime)</code>	Make duration instance

Method signature	Description
<code>makeBase64BinaryInstance(byte[])</code>	Make base 64 binary instance from byte array
<code>makeBase64BinaryInstance(gw.xml.BinaryDataProvider)</code>	Make base 64 binary instance from binary data provider

XSD to Gosu Simple Type Mappings

For all elements with simple types and all attributes in an XSD, Gosu creates properties based on which simple schema type it is. The following table describes how Gosu maps XSD schema types to Gosu types. For schema types that are not listed in the table, Gosu uses the schema type's supertype. For example, the schema type `String` is not listed, so Gosu uses its supertype `anySimpleType`.

Schema Type	Gosu Type
<code>boolean</code>	<code>java.lang.Boolean</code>
<code>byte</code>	<code>java.lang.Byte</code>
<code>decimal</code>	<code>java.math.BigDecimal</code>
<code>double</code>	<code>java.lang.Double</code>
<code>float</code>	<code>java.lang.Float</code>
<code>int</code>	<code>java.lang.Integer</code>
<code>integer</code>	<code>java.math.BigInteger</code>
<code>long</code>	<code>java.lang.Long</code>
<code>short</code>	<code>java.lang.Short</code>
<code>unsignedLong</code>	<code>java.math.BigInteger</code>
<code>unsignedInt</code>	<code>java.lang.Long</code>
<code>unsignedShort</code>	<code>java.lang.Integer</code>
<code>unsignedByte</code>	<code>java.lang.Short</code>
<code>date</code>	<code>gw.xml.date.XmlDate</code>
<code>dateTime</code>	<code>gw.xml.date.XmlDateTime</code>
<code>time</code>	<code>gw.xml.date.XmlTime</code>
<code>gYearMonth</code>	<code>gw.xml.date.XmlYearMonth</code>
<code>gYear</code>	<code>gw.xml.date.XmlYear</code>
<code>gMonthDay</code>	<code>gw.xml.date.XmlMonthDay</code>
<code>gDay</code>	<code>gw.xml.date.XmlDay</code>
<code>gMonth</code>	<code>gw.xml.date.XmlMonth</code>
<code>duration</code>	<code>gw.xml.date.XmlDuration</code>
<code>base64Binary</code>	<code>gw.xml.BinaryDataProvider</code>
<code>hexBinary</code>	<code>byte[]</code>
<code>anyURI</code>	<code>java.net.URI</code>
<code>QName</code>	<code>javax.xml.namespace.QName</code>
<code>IDREF</code>	<code>gw.xml.XmlElement</code>
<code>anySimpleType</code>	<code>java.lang.String</code>
any type with enumeration facets	schema-specific enumeration type
any type derived by list of T	<code>java.util.List<T></code>
any type derived by union of (T1, T2,... Tn)	greatest common supertype of (T1, T2,... Tn)

Facet Validation

A facet is a characteristic of a data type that restricts possible values. For example, setting a minimum value or matching a specific regular expression.

Gosu represents each facet as an element. Each facet element has a fixed attribute that is a Boolean value. All the facets for a simple type collectively define the set of legal values for that simple type.

Most schema facets are validated at property setter time. A few facets are not validated until serialization time to allow incremental construction of lists at runtime. This mostly affects facets relating to lengths of lists, and those that validate QName objects. Gosu cannot validate QName objects at property setting time because there is not enough information available. Also, the XML Schema specification recommends against applying facets to QName objects at all.

Example XSD

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
    <xsd:complexType>
      <xsd:attribute name="Attr1" type="AttrType"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:simpleType name="AttrType">
    <xsd:restriction base="xsd:int">
      <xsd:minInclusive value="0"/>
      <xsd:maxInclusive value="5"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Code

```
var xml = new schema.Element1()
xml.Attr1 = 3 // Works!
xml.Attr1 = 6 // Fails with an exception exception.
```

Output

```
gw.xml.XmlSimpleValueException: Value '6' violated one or more facet constraints
of simple type definition: value must be no greater than 5
```

Access the Nillness of an Element

XML has a concept of whether an element is `nil`. This is not exactly same as being `null`. An element can be `nil` (and must have no child elements) but still have attributes. Additionally, an XSD can define whether an element is *nillable*, which means that element is allowed to be `nil`.

If an XSD-based element is nillable, the `XmLElement` object exposes a property with the name `$Nil`. All non-XSD elements also have this property, but it is called `Nil` (with no dollar sign prefix). Nillability is an XSD concept, so for non-XSD elements the element always potentially can be `nil`.

Note: For XSD-based elements not marked as nillable, this property is unsupported. In the Gosu editor, if you attempt to use the `$Nil` property, Gosu generates a deprecation warning.

Setting this property on an element to `true` affects whether upon serialization Gosu adds an `xsi:nil` attribute on the element. Getting this property returns the state of that flag (`true` or `false`).

Nillability is an aspect of XSD-based elements, not an aspect of the XSD type itself.

Example XSD

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1" type="xsd:int" nillable="true"/>
</xsd:schema>
```

Code

```
var xml = new schema.Element1()
xml.$Nil = true
xml.print()
```

Output

```
<?xml version="1.0"?>
<Element1 xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
```

For more on the distinction between `XmlElement` and its backing type, see “Introduction to the XML Element in Gosu” on page 271.

Automatic Creation of Intermediary Elements

If you use XSDs, whenever a property path appears in the left hand side of an assignment statement, Gosu creates any intermediary elements to assure the assignment from Gosu works. This is a very useful shortcut. Use this feature to make your Gosu code significantly more understandable.

Example XSD

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Child1">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Child2" type="xsd:int"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Code

```
var xml = new schema.Element1()
print("Before assignment: ${xml.Child1}")
xml.Child1.Child2 = 5 // Assignment of a value to Child2 automatically creates Child1.
print("After assignment: ${xml.Child1}")
```

Output

```
Before assignment: null
After assignment: schema.anonymous.elements.Element1_Child1 instance
```

Default and Fixed Attribute Values

The default values for `default` and `fixed` attributes and elements come from the statically typed property getter for those attributes and elements. These default values are not stored in the attribute map or content list for a an XML type. Gosu adds default or fixed values to attributes and elements in the XML output stream at the time that Gosu serializes the Gosu representation of an XML document.

Example XSD

The following example XSD defines an XML element named `person`. The element definition includes an attribute definition named `os` with a default value of “Windows” and an attribute definition named `location` with a fixed value of “San Mateo”.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="root">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="person" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="name" type="xsd:string"/>
            </xsd:sequence>
            <xsd:attribute name="os" type="ostype" default="Windows"/>
            <xsd:attribute name="location" type="xsd:string" fixed="San Mateo"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

        </xsd:complexType>
    </xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:simpleType name="ostype">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Windows"/>
        <xsd:enumeration value="MacOSX"/>
        <xsd:enumeration value="Linux"/>
    </xsd:restriction>
</xsd:simpleType>
</xsd:schema>

```

Code

The following sample Gosu code creates a new Gosu representation of an XML document based on the preceding XSD. The code adds two `person` elements, one for `jsmith` and one for `aanderson`. For `jsmith`, the code adds an `os` attribute set to the value `Linux`. The code does not add an `os` attribute to `aanderson`, nor does the code add the `location` attribute to either person. Instead, the code relies on the default and fixed values defined in the XSD.

```

var xml = new schema.Root()
xml.Person[0].Name = "jsmith"
xml.Person[0].Os = Linux
xml.Person[1].Name = "aanderson"

// Gosu adds default and fixed values to the XML document at the time Gosu serializes XML for print.
for (person in xml.Person) {
    print("${person.Name} (${person.Location}) -> ${person.Os}")
}

xml.print()

```

Output

At the time the preceding Gosu code serializes its representation of an XML document, Gosu adds the fixed and default values to the XML output stream. The printed output shows that the Gosu representation of the XML document does not contain the value `San Mateo` or `Windows`.

```

jsmith (San Mateo) -> Linux
aanderson (San Mateo) -> Windows
<?xml version="1.0"?>
<root>
    <person os="Linux">
        <name>jsmith</name>
    </person>
    <person>
        <name>aanderson</name>
    </person>
</root>

```

Substitution Group Hierarchies

Just as Gosu reproduces XSD-defined type hierarchies in the Gosu type system, Gosu also exposes XSD-defined substitution group hierarchies.

The name *substitution group* is the standard name for this XSD feature, although the name can be somewhat confusing. You can define an XSD `substitutionGroup` attribute on any top-level element to indicate the QName of another top-level element for it can substitute.

The name *substitution group* comes from its normal use, which is to create a substitution group head (the group's main element) with some abstract name, such as "Address".

To create a substitution group member, set the XML attribute `substitutionGroup` on an element to the element name (QName) of the substitution group head.

There is no need to indicate at runtime that the substitution happened place. This is in contrast to subtypes, in which `xsi:type` must be present. If an XML element uses a substitution group member QName in place of the head's QName, the Gosu XML processor knows that the substitution happened.

Example XSD

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Address"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Address"/>
  <xsd:element name="USAddress" substitutionGroup="Address"/>
  <xsd:element name="UKAddress" substitutionGroup="Address"/>
</xsd:schema>
```

Code

```
var xml = new schema.Customer()
xml.Address = new schema.UKAddress()
xml.print()
```

Output

```
<?xml version="1.0"?>
<Customer>
  <UKAddress/>
</Customer>
```

The XML Schema specification requires that the XSD type of a substitution group member must be a subtype of the XSD type of its substitution group head. The reason the example above works is because `UKAddress`, `USAddress` and `Address` are all of the type `xsd:anyType` (the default when there is no explicit type).

Element Sorting for XSD-based Elements

An XSD can define the strict order of children of an element. For non-XSD elements, element order is undefined.

Each `Xmlelement` exposes a `Children` property. (For XSD-based elements the property name is `$Children`.)

If the list of child elements is out of order according to the XSD, Gosu sorts the element list during serialization to match the schema. This sorting does not affect the original order of the elements in the content list.

If you use APIs to directly add child elements, such as adding to the child element list or using an `addChild` method, you can add child elements out of order. Similarly, some APIs indirectly add child elements, such as such as autocreation of intermediary elements (see “Automatic Creation of Intermediary Elements” on page 295.). In all of these cases, Gosu permits the children to be out of order in the `Xmlelement` object graph.

During serialization and only during serialization, Gosu sorts the elements to ensure that the elements conform to the XSD.

Note that if you parse XML into an `Xmlelement` using an XSD, the elements must be in the correct order according to the XSD. If the child order violates the XSD, Gosu throws an exception during parsing.

Example XSD

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Child1" type="xsd:int"/>
        <xsd:element name="Child2" type="xsd:int"/>
        <xsd:element name="Child3" type="xsd:int"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Code

```
var xml = new schema.Element1()
xml.Child2 = 2
xml.Child1 = 1
xml.Child3 = 3
xml.print()
```

Output

```
<?xml version="1.0"?>
<Element1>
  <Child1>1</Child1>
  <Child2>2</Child2>
  <Child3>3</Child3>
</Element1>
```

Example XSD

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
    <xsd:complexType>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element name="A" type="xsd:int"/>
          <xsd:element name="B" type="xsd:int"/>
          <xsd:element name="C" type="xsd:int"/>
        </xsd:sequence>
        <xsd:sequence>
          <xsd:element name="B" type="xsd:int"/>
          <xsd:element name="C" type="xsd:int"/>
          <xsd:element name="A" type="xsd:int"/>
          <xsd:element name="Q" type="xsd:int"/>
        </xsd:sequence>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Code

```
var xml = new schema.Element1()
xml.A = 5
xml.B = 5
xml.C = 5
xml.print()
print( "-----" )
xml.Q = 5
xml.print()
```

Output

```
<?xml version="1.0"?>
<Element1>
  <A>5</A>
  <B>5</B>
  <C>5</C>
</Element1>
-----
<?xml version="1.0"?>
<Element1>
  <B>5</B>
  <C>5</C>
  <A>5</A>
  <Q>5</Q>
</Element1>
```

If Element Order Is Already Correct

If the children of an element are in an order that matches the XSD, Gosu does not sort the element list. This is important if there is more than one sorted order that conforms to the XSD and you desire a particular order.

Example XSD

The following XSD defines two distinct strict orderings of the same elements:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
```

```
<xsd:complexType>
  <xsd:choice>
    <xsd:sequence>
      <xsd:element name="A" type="xsd:int"/>
      <xsd:element name="B" type="xsd:int"/>
      <xsd:element name="C" type="xsd:int"/>
    </xsd:sequence>
    <xsd:sequence>
      <xsd:element name="C" type="xsd:int"/>
      <xsd:element name="B" type="xsd:int"/>
      <xsd:element name="A" type="xsd:int"/>
    </xsd:sequence>
  </xsd:choice>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Code

```
var xml = new schema.Element1()
xml.A = 5
xml.B = 5
xml.C = 5
xml.print()

print( "-----" )

xml = new schema.Element1()
xml.C = 5
xml.B = 5
xml.A = 5
xml.print()
```

Output

```
<?xml version="1.0"?>
<Element1>
  <A>5</A>
  <B>5</B>
  <C>5</C>
</Element1>
-----
<?xml version="1.0"?>
<Element1>
  <C>5</C>
  <B>5</B>
  <A>5</A>
</Element1>
```

Multiple Correct Sort Order Matches

If the children of an element are out of order, but multiple correct orderings exist, the first correct ordering defined in the schema will be used.

Example XSD

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
    <xsd:complexType>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element name="A" type="xsd:int"/>
          <xsd:element name="B" type="xsd:int"/>
          <xsd:element name="C" type="xsd:int"/>
        </xsd:sequence>
        <xsd:sequence>
          <xsd:element name="C" type="xsd:int"/>
          <xsd:element name="B" type="xsd:int"/>
          <xsd:element name="A" type="xsd:int"/>
        </xsd:sequence>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Code

```
var xml = new schema.Element1()
xml.C = 5
xml.A = 5
xml.B = 5
xml.print()
```

Output

```
<?xml version="1.0"?>
<Element1>
  <A>5</A>
  <B>5</B>
  <C>5</C>
</Element1>
```

Built-in Schemas

Gosu includes several XSDs in the `gw.xsd.*` package. The following table lists the built-in XSDs.

Description of the XSD	Fully-qualified XSD package name
The SOAP XSD	<code>gw.xsd.w3c.soap</code>
SOAP envelope XSD	<code>gw.xsd.w3c.soap_envelope</code>
WSDL XSD	<code>gw.xsd.w3c.wsdl</code>
XLink XSD (for linking constructs)	<code>gw.xsd.w3c.xlink</code>
The XML XSD, which defines the attributes that begin with the <code>xml:</code> prefix, such as <code>xml:lang</code> .	<code>gw.xsd.w3c.xml</code>
XML Schema XSD, which is the XSD that defines the format of an XSD. See “The XSD that Defines an XSD (The Metaschema)” on page 300.	<code>gw.xsd.w3c.xmlschema.Schema</code>

The XSD that Defines an XSD (The Metaschema)

The definition of an XSD is itself an XML file. The *XML Schema* XSD is the XSD that defines the XSD format. It is also known as the *metaschema*. It is in the Gosu package `gw.xsd.w3c.xmlschema`. This schema is useful sometimes for building or parsing schemas.

Example Code

```
var schema = new gw.xsd.w3c.xmlschema.Schema()
schema.Element[0].Name = "Element1"
schema.Element[0].ComplexType.Sequence.Element[0].Name = "Child"
schema.Element[0].ComplexType.Sequence.Element[0].Type = new javax.xml.namespace.QName( "Type1" )
schema.ComplexType[0].Name = "Type1"
schema.print()
```

Output

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Child" type="Type1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="Type1"/>
</xsd:schema>
```

There is no way to inject a schema into the type system at run time.

Use a Local XSD for an External Namespace or XSD Location

Sometimes an XSD refers to another XSD that is external. It is strongly discouraged to require XSD processing to connect across the internet to access that XSD. Network connections are low performance and unreliable in a production system, and the external XSD might not be reachable. Instead, it is best to copy the XSD locally to the resource tree and tell Gosu how to map the external XSD namespace to a local XSD.

In a slightly different situation, it is common for an XSD to define an external namespace URI without necessarily specifying the external download location for an XSD.

In both cases, Gosu provides a registry file that lets you use a local XSD instead.

To use a local XSD for an external namespace URI or external XSD location URL

1. Get a copy of the target external XSD.
2. Place that XSD in the local resources hierarchy.
3. Modify the file `schemalocations.xml` in Studio.

It looks like this:

```
<?xml version="1.0"?>
<!-- XML Namespaces and external schema locations can be registered in this file to
map to local resources by default.
These defaults are used when an <xsd:import> is encountered without a schemaLocation
specified, or an external
(non-local) schema location is specified. The "externalLocations" attribute is
a space-delimited list of
external references to recognize for a particular schema. -->
<schemalocations xmlns="http://guidewire.com/xml/schemalocations">
  <schema xmlns="http://guidewire.com/archiving"
    resourcePath="gw/plugin/archiving/archiving.xsd" externalLocations="" />
  <schema xmlns="http://guidewire.com/importing"
    resourcePath="gw/api/importing/importing.xsd" externalLocations="" />
</schemalocations>
```

4. Add a new `<schema>` element.
5. Set the `resourcePath` attribute to the fully-qualified path to the local XSD. It is important to use a forward slash instead of a period to indicate any subpackage.
6. If you want Gosu to find and remap an external namespace declaration in parsed XSD, set the `xmlns` attribute to the external XSD namespace URI. This kind of value looks like a quoted URL, and starts with `http://` but is actually a unique ID not a download URL.
7. If you want Gosu to find and remap an external XSD download location in parsed XSD, set the `externalLocations` attribute to the external XSD download URL. This may also be useful for your own reference, in case you need to download the latest version of the file again.

Be aware there are multiple versions of the file `schemalocations.xml` in the default configuration. Each one is in a different module and actually Gosu uses all of them. When you edit `schemalocations.xml` in Studio, you are only editing the application-level copy. There are other versions in other modules that are actually used, though not visible. Other versions of this XML file remap the common W3C XSDs to local locations to avoid attempting to connect to the W3C web site.

Also see “Referencing Additional Schemas During Parsing” on page 279.

Schema Access Type

For each XSD that Gosu loads, it creates a `SchemaAccess` object that represents the loaded XSD. The most important reason to load XSDs is to provide Gosu with additional schemas during XML parsing. Additionally, `SchemaAccess` objects have a `Schema` property, which is the Gosu XML representation of the XSD. In technical terms, the `Schema` property contains the `gw.xsd.w3c.xmlschema.Schema` object that represents the XSD.

See also

- “Parsing XML Data into an XML Element” on page 278
- “The XSD that Defines an XSD (The Metaschema)” on page 300

Example XSD

Suppose you have this XSD loaded as `schema.util.SchemaAccess.Schema`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1"/>
  <xsd:element name="Element2"/>
  <xsd:element name="Element3"/>
</xsd:schema>
```

Code

```
var schema = schema.util.SchemaAccess.Schema
schema.Element.each(\ el ->print(el.Name))
```

Output

```
Element1
Element2
Element3
```

Code

The following example uses the XSD of XSDs to print a list of primitive schema types:

```
var schema = gw.xsd.w3c.xmlschema.util.SchemaAccess.Schema
print(schema.SimpleType.where(\ s ->s.Restriction.Base.LocalPart == "anySimpleType").map(
  \ s ->s.Name))
```

Output

```
[string, boolean, float, double, decimal, duration, dateTime, time, date, gYearMonth,
gYear, gMonthDay, gDay, gMonth, hexBinary, base64Binary, anyURI, QName, NOTATION]
```

The Guidewire XML (GX) Modeler

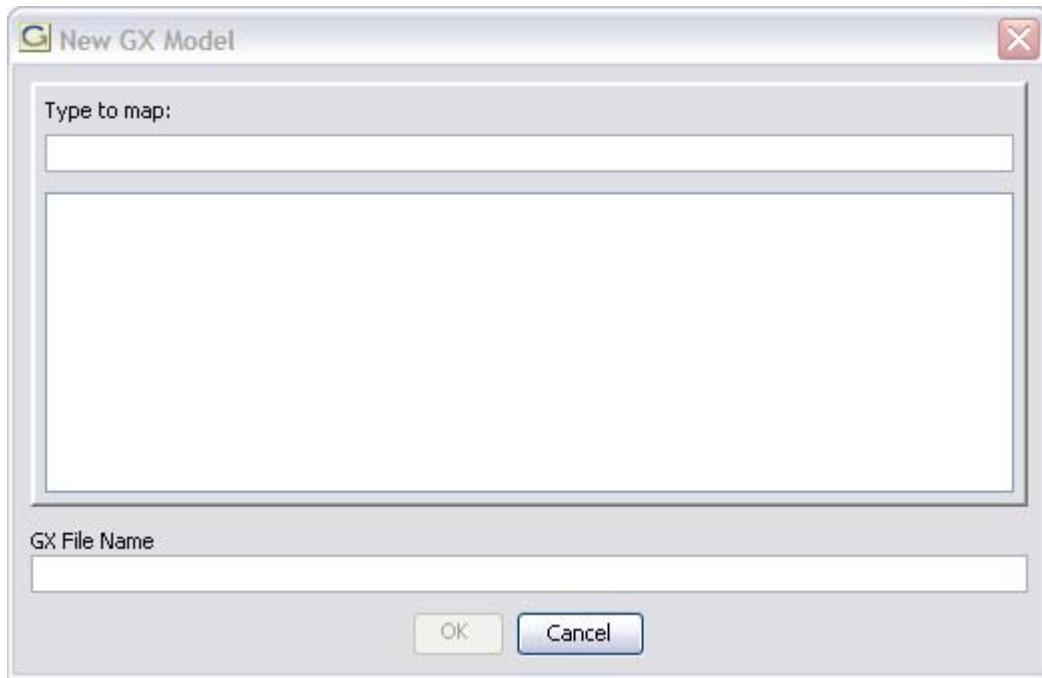
You can export business data entities, Gosu class data, and other types to a standard Guidewire XML format. You can select which properties to map in your XML model. By specifying what to map, ClaimCenter creates a XSD to describe XML that conforms to your XML model. At run time, you can export XML for 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** → **gsrc**. 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 **gsrc**, and select **New Package**. Next, right-click on the desired package. From the contextual menu, choose **New** → **GX Model**.

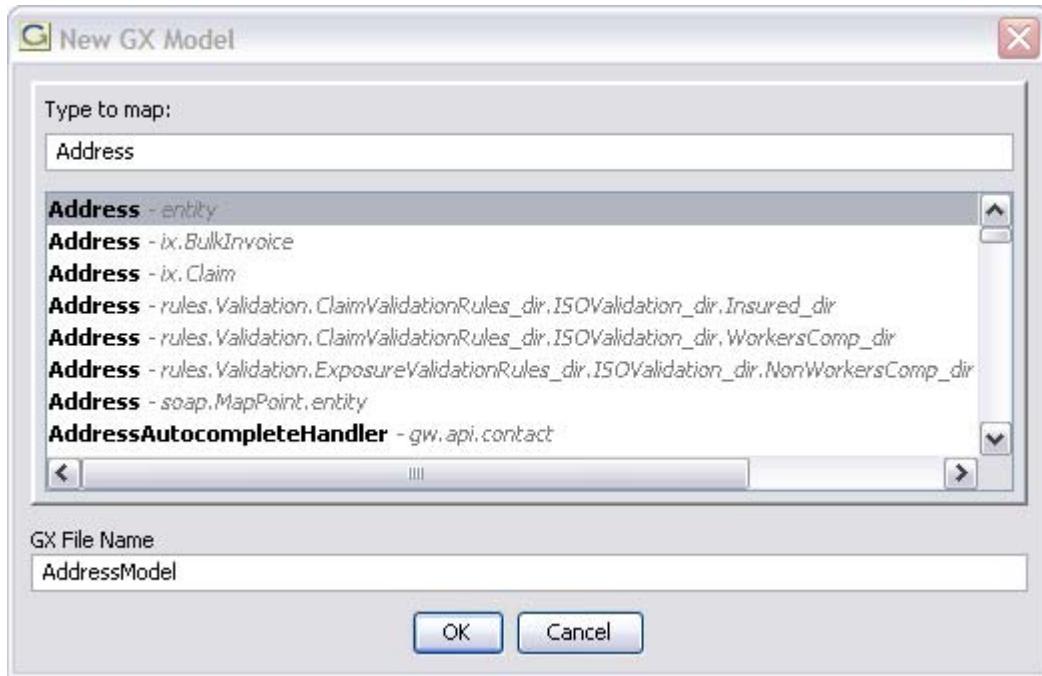
Sometimes in code or in other documentation, you might notice the Guidewire XML Model abbreviated in documentation as GX. GX is a shortened form of the name Guidewire XML. For example, GX models and the GX modeler tool.

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.

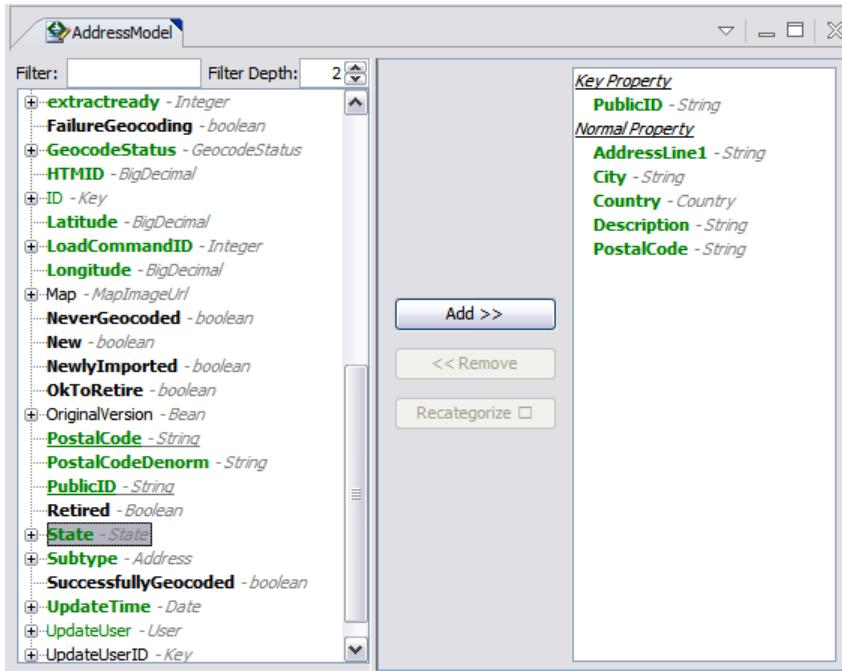
It is common for the list to include multiple types with identical names but different packages (namespaces) listed in gray. Be sure to choose the correct type in the list during this step.



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 `AddressModel1`, the fully-qualified type for the `Address` entity in your type model is `mycompany.messaging.AddressModel1.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, see “Gosu Type to XSD Type Conversion Reference” on page 326.
 - 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 editor 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, which are 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 227)

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 307.

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 307.

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.

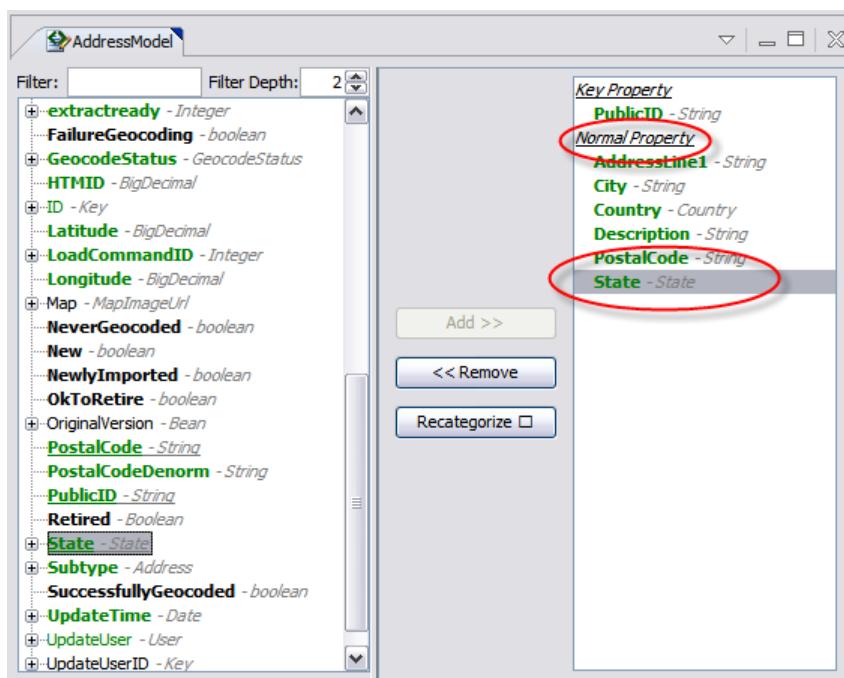
If you do not want to navigate through the entire hierarchy, you can instead use the search box to find the desired property by name. Studio updates the tree view to include properties with the name you typed if it is within the depth level from the root of the type. For example, suppose you have a type 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/ws
```

The web server displays 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 `AddressModel1`. 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.AddressModel1.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 `Bytes` property (the preferred method) and the `asUTFString` method.

The examples in this section by default generate the entire XML graph. If you use a Guidewire Standard XML model, you can choose additional export options:

- `Verbose` – Export elements even for `null` properties using the `Verbose` property in `GXOptions`.
- `Incremental` – Export only entity properties that changed using the `Incremental` property in `GXOptions`.

See also

- “Customizing GX Modeler Output (`GXOptions`)” on page 308
- “Legacy XML APIs: Exporting XML Data” on page 316
- “The Guidewire XML (GX) Modeler” on page 302

Customizing GX Modeler 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.

GXOptions 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 <code>false</code> .	If set to <code>true</code> , 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 <code>false</code> , generates a property's XML even if it did not change.	No effect. The incremental feature does not apply to this type of property.

GXOptions Property	Description	Behavior for properties backed by database columns	Behavior for properties not backed by database columns
Verbose	<p>Export an XML element for an property even if the value for that property is null</p> <p>The default for Verbose is false. This is the behavior if you use the one-argument constructor without the GXOptions object.</p>	<p>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 sub-elements in the XML. For example, the following XML represents the property mytype with a value null:</p> <pre><mytype xsi:nil="true"></pre> <p>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.</p> <p>If the Verbose option is false, a null value does not export an element in XML.</p>	<p>Same behavior as for database columns. See previous table cell.</p>
SupressExceptions	<p>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 311. The default is false</p>	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 entities inside the virtual property graph changed.

For example in ClaimCenter, the primary address on a claim is a virtual property on the Claim entity. Any subobjects of Claim.PrimaryAddress cannot be tested for new, deleted, or 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. The following example sets both `Incremental` and `Verbose` to true:

```
var xml = new mycompany.messaging.AddressModel.Address(myAddress,
    new GXOptions() { :Incremental = true , :Verbose=true })
```

For details of data structure initialization syntax, see “Object Initializer Syntax” on page 75.

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-data-model 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-data-model 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-data-model 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 `SuppressExceptions`.

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.

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.

For details of the `Incremental` option, see “Generating XML Using an XML Model” on page 307.

Special Behavior for Entity Arrays Inside Virtual Property Graphs

The formatting behavior is slightly different for 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 the `Incremental` option were set to `false`.

For behaviors that differ for entities inside virtual property graphs, see “Generating XML Using an XML Model” on page 307 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 that 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` – Type `xsd:string`

- Hour – Type xsd:int
- Minute – Type xsd:int

9. Open the Gosu Scratchpad 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 Scratchpad:

```
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 Scratchpad in Studio:

```
var sampleXML = "<Date xmlns='http://guidewire.com/gx/gxdemo.DateModel'>" +
  "<DayOfWeekName>Friday</DayOfWeekName>" +
  "</Date>"
var xml = gxdemo.DateModel.Date.parse(sampleXML)
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.

XML Serialization Performance and Element Sorting

For large XML objects with many nested layers, sorting requires a lot of computer resources. See “Serialization Performance and Element Sorting” on page 277.

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 326.

Legacy XML APIs: Manipulating XML as Untyped Nodes

Note: This section applies only to the legacy XML APIs that use the XMLNode class.

To load XML into the native Gosu parser

- 1. Get XML as String, File, or Input Stream** – 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)
```

As soon as 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 one based on initial XML, create the initial node with the new element approach.

See also

- “New Elements” on page 314
- “Differences Between Legacy XML APIs and Current XML APIs” on page 269

Untyped Node Operations

As soon as 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 )

// Get text contents from an element.
var t = root.Children[0].Children[0].text
print ("contents are " + t)

// Set 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 descendants 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 “Legacy XML APIs: Exporting XML Data” on page 316.

Example of Manipulating XML as Untyped Nodes

The example code below demonstrates the following steps of logic:

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 Scratchpad 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 from the list root.Children.
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>

```

Legacy XML APIs: Exporting XML Data

Note: This section applies only to the legacy XML APIs that use the `XMLNode` class. For more information about the differences between the current XML API and the legacy XML API, see “[Differences Between Legacy XML APIs and Current XML APIs](#)” on page 269.

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 sends or stores only 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, see

<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, see

<http://tools.ietf.org/html/rfc3629>

IMPORTANT Remember to test your receiving system to assure it properly decodes UTF-8 text. Be careful to test your integration code with non-English characters, which have 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, optionally you can choose other export options, such as:

- `Verbose` – Export elements even for `null` properties using the `Verbose` property in `GXOptions`.
- `Incremental` – Export only entity fields that changed using the `Incremental` property in `GXOptions`.

See also

- “Customizing GX Modeler Output (`GXOptions`)” on page 308
- “The Guidewire XML (GX) Modeler” on page 302

Comments on an XML Node

During XML parsing, Gosu imports comments associated with each element (each `XMLNode`). You can read or write the `node.Comment` property. During serialization of a node, Gosu includes the comment.

Legacy XML APIs: Collection-like Enhancements for XML

Note: This section applies only to the legacy XML APIs that use the `XMLNode` class. For more information about the differences between the current XML API and the legacy XML API, see “Differences Between Legacy XML APIs and Current XML APIs” on page 269

Given an XML node, you can easily extract a list of direct child nodes (child elements), or a list of all nodes (all descendant elements) in the tree. In both cases, Gosu returns the nodes as a `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 239. 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 251. For more information about blocks, see “Gosu Blocks” on page 231.

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 251. 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. See 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 251 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
```

Legacy XML APIs: Structured XML Using XSDs

Note: This section applies only to the legacy XML APIs that use the `XMLNode` class. For more information about the differences between the current XML API and the legacy XML API, see “Differences Between Legacy XML APIs and Current XML APIs” on page 269

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 “Legacy XML APIs: Manipulating XML as Untyped Nodes” on page 313. 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 package hierarchy defines the package into which Gosu loads your XSD. In other words, where you put the XSD defines the XSD package and the all related entities that load into Gosu from your XSD.

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.*`.

Next you must tell ClaimCenter import an XSD into the Gosu type system using the legacy XML system. To do this, copy the file `ClaimCenter/modules/pl/config/registry/compatibility-xsd.xml` into the configuration module at the same path. Next, add an entry for your XSD. Set the `namespace` attribute to the Gosu package name of the schema. For example, if the schema is in the package location `example.pkg` and is called `myschema.xsd`, set the namespace to `example.pkg.myschema`.

IMPORTANT To load an XSD in Gosu using the legacy XSD loading system, you must add elements to the `compatibility-xsd.xml` file.

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 “Legacy XML APIs: Collection-like Enhancements for XML” on page 318.

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 XSD Choices in XML” on page 324 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 defined 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 package 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 322.
- 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 Studio 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 “Legacy XML APIs: Manipulating XML as Untyped Nodes” on page 313.

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 “Legacy XML APIs: Collection-like Enhancements for XML” on page 318.

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. Throws 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.

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()
```

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, Gosu sometimes cannot determine the element name until after a node is assigned to a parent. At that point, the structure of the XSD defines which element the node represents.

For more details of XML export and issues related to Unicode and Guidewire messaging, see “Legacy XML APIs: Exporting XML Data” on page 316.

Handling XSD Choices in XML

There are several things to know about XSD choices, which constrain XML documents to contain only one of several different elements.

For example, the following is a simple XSD that contains 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)

// Print the work phone, but which phone number is now 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`.

Also see “What Types Does Gosu Generate?” on page 321

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
<code>xsd:ENTITIES</code>	<code>java.lang.String</code>
<code>xsd:ENTITY</code>	<code>java.lang.String</code>
<code>xsd:ID</code>	<code>java.lang.String</code>
<code>xsd:IDREF</code>	<code>gw.xml.xsd.IXMLNodeWithID<gw.xml.IReadOnlyXMLNode></code>
<code>xsd:IDREFS</code>	<code>java.util.List<gw.xml.xsd.IXMLNodeWithID<gw.xml.IReadOnlyXMLNode>></code>
<code>xsd:NCName</code>	<code>java.lang.String</code>
<code>xsd:NMTOKEN</code>	<code>java.lang.String</code>

XSD type in custom XSD	Maps to this Gosu type
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: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 302.

Gosu type	Guidewire XML Model maps to this XSD type
boolean	xsd:boolean
byte	xsd:byte

Gosu type	Guidewire XML Model maps to this XSD type
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
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 Qualified Names

The XML parser supports XSD qualified names (QName objects) with the xsd:QName XSD type. The XSD standard uses QName objects 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.namespace.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 in boldface. 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="F001" categoryName="category #1"/>
  <category categoryId="F002" categoryName="category #2"/>
  <item categoryRef="F001" 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 Scratchpad:

```
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 it to the category objects.
var category = xml.items[0].categoryRef as gw.mytest.category

// Now you 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.

Bundles and Database Transactions

Gosu provides APIs to change how changes to Guidewire entity data save to the database. For many programming tasks in Gosu, such as typical rule set code, you may not need to know how database transactions work. For some situations, however, you must understand database transactions, for example:

- adding entities to the current database transaction
- moving entities from one database transaction to another
- explicitly saving data to the database (committing a bundle)
- undo the current database transaction by throwing Gosu exceptions.

WARNING Only commit entity changes at appropriate times or you could cause serious data integrity issues. In many cases (such as typical Rule sets and PCF code), it is best to rely on default behavior and do not explicitly commit entity data manually.

This topic includes:

- “When to Use Database Transaction APIs” on page 332
- “Bundle Overview” on page 334
- “Adding Entity Instances to Bundles” on page 334
- “Getting the Bundle of an Existing Entity Instance” on page 335
- “Getting an Entity from a Public ID or a Key (Internal ID)” on page 336
- “Creating New Entity Instances in Specific Bundles” on page 336
- “Committing a Bundle Explicitly in Very Rare Cases” on page 337
- “Removing Entity Instances from the Database” on page 337
- “Determining What Data Changed in a Bundle” on page 338
- “Running Code in an Entirely New Bundle” on page 342
- “Exception Handling And Database Commits” on page 344
- “Bundles and Published Web Services” on page 344
- “Entity Cache Versioning, Locking, and Entity Refreshing” on page 344
- “Details of Bundle Commit Steps” on page 346

When to Use Database Transaction APIs

Gosu provides APIs that change how to save business data object changes to the database. ClaimCenter groups all related entity instance changes together in a group called a *bundle*. There are APIs to manipulate entity data in bundles.

For many programming tasks in Gosu, you do not need to know details of how database transactions work. For example, after rule set execution, the application sends to the database all related changes to current objects. This is called *committing* the bundle.

If errors occur that prevent the entire commit action from safely completing, no data changes commit to the database. The bundle is not inherently destroyed. Depending on application context, the bundle might be preserved, for example to let a user fix validation errors. In contrast, in a web service implementation, a commit fail typically results in discarding the temporary bundle and returning an error to the web service client.

The most important types of database transaction APIs are:

- “Adding Entity Instances to Bundles” on page 334
- “Running Code in an Entirely New Bundle” on page 342
- “Committing a Bundle Explicitly in Very Rare Cases” on page 337

WARNING Using database transaction APIs has tremendous effects on application logic and data integrity. Using APIs incorrectly can adversely affect other application logic that tracks when to commit changes or undo recent data changes.

The following table lists typical requirements for using the main database transaction APIs.

Code context	Create a new bundle with <code>runWithNewBundle?</code>	Add entity instances to bundles before changing data?	Commit bundle explicitly for data changes?	Notes
Code that only reads properties	No	No	No	If you do not change entity data, you do not need to use any database transaction APIs.
Rule sets	No	Yes, for database query results. See note.	No	<p>It is dangerous and unsupported for rule set code to explicitly commit any bundle.</p> <p>Some rule sets have a main object that is already in the current writable bundle. For example validation rules or pre-setup rules.</p> <p>If you change only that main entity and its subobjects, your Gosu rule set code does not typically require special database transaction APIs.</p>

Code context	Create a new bundle with <code>runWithNewBundle?</code>	Add entity instances to bundles before changing data?	Commit bundle explicitly for data changes?	Notes
PCF code in: <ul style="list-style-type: none"> • list views, • view screens • edit screens 	No	Yes, for database query results.	No	<p>Typical PCF widgets handle database transactions for you. For example, after a user enters Edit mode, the application creates a new bundle. After the user clicks the Save button, the application commits the bundle with the user changes. In general, use the default PCF behaviors.</p>
PCF pages with asynchronous actions	Yes	Yes, for database query results.	No. However, you use the <code>runWithNewBundle</code> API, which commits the bundle for you.	<p>In wizard user interface flow, you can optionally configure certain steps to commit the latest changes. However, even in those cases you are not explicitly committing a bundle.</p>
Plugin implementation code	No	Yes, for database query results. Plugin method arguments that are intended to be modified are in a current bundle that is writable.	No	<p>Before the application calls plugin methods, the application sets up a current bundle.</p>
Workflow code	No	Yes, for database query results	No.	<p>The current bundle for workflow actions is the bundle that the application uses to load the <code>Workflow</code> entity instance.</p>
Web services (WS-I type)	Yes, if the operation modifies data.	Yes, for database query results	No. However, you use the <code>runWithNewBundle</code> API, which commits the bundle for you.	<p>For WS-I, see “Running Code in an Entirely New Bundle” on page 342. The API discussed in that section automatically commits the bundle. You do not explicitly need to commit the bundle.</p>
Web services (RPCE type)	It depends. If you only modified method arguments, it is unnecessary to create a new bundle. If you modify other data or only want to commit some arguments but not others, then no.	Yes, for database query results	Yes	<p>Implementations of RPCE web services are the extremely rare exception in which you need to actually commit a database transaction explicitly.</p> <p>WARNING: RPCE web services are deprecated. Convert existing RPCE code to WS-I web services.</p>

Bundle Overview

To manage database transactions, Guidewire applications group entity instances in groups called *bundles*. A bundle is a collection of in-memory entity instances that represent rows in the database. The application transmits and saves all entity instances in the bundle to the database in one transaction. A bundle includes changed entities, new entities, and entities to delete from the database. Gosu represents a bundle with the class `gw.transaction.Bundle`.

Guidewire refers to the process of sending the entities to the database as *committing the bundle* to the database. If a bundle commit attempt completely succeeds, all database changes happen in a single database transaction. If the commit attempt fails in any way, the entire update fails and Gosu throws an exception.

There are two basic types of bundles: *read-only* bundles and *writable* bundles. If you do a database query, the results of the query are in a temporary read-only bundle and you must copy it to a writable bundle to change any data. See “[Making an Entity Instance Writable By Adding to a Bundle](#)” on page 334.

Not all writable bundles eventually commit to the database. For example:

- A user might start to make data changes in the user interface but abandon the task.
- A user might start to make data changes in the user interface, try to save them, but errors prevent completion. Eventually, the user might cancel the action before fixing and completing the action.
- A batch process might attempt a database change, but errors prevent completing the action.
- A web service call might attempt a database change, but errors prevent completing the action.

If any code destroys a bundle with uncommitted changes, it is as if the changes never occurred. All entity data in the database is unchanged.

The following topics describe common actions that use existing bundles:

- “[Adding Entity Instances to Bundles](#)” on page 334
- “[Getting the Bundle of an Existing Entity Instance](#)” on page 335
- “[Getting an Entity from a Public ID or a Key \(Internal ID\)](#)” on page 336
- “[Creating New Entity Instances in Specific Bundles](#)” on page 336
- “[Committing a Bundle Explicitly in Very Rare Cases](#)” on page 337
- “[Determining What Data Changed in a Bundle](#)” on page 338

Adding Entity Instances to Bundles

Making an Entity Instance Writable By Adding to a Bundle

It is common to query the database for specific entity instances and then make changes to the data. See “[Overview of the Query Builder APIs](#)” on page 125. However, the direct results of a database query are in a temporary read-only bundle. You must copy objects to a writable bundle to change objects or delete objects. In most programming contexts, there is a current bundle that the application has already prepared. For example, in all rule execution contexts, and typical PCF user interface code. When there is a current bundle, get the current bundle using the code:

```
gw.transaction.Transaction.getCurrent()
```

Note: In batch processes, WS-I web service implementations, and in workflows, there is no current bundle. You need to create a new bundle for your data changes. See “[Running Code in an Entirely New Bundle](#)” on page 342.

To add an entity instance to a bundle, pass the object reference to the `bundle.add(obj)` method and save the return result. It is extremely important to save the return value of this method. The return result of the `add` method is a copy of the object that exists inside the new bundle. Only use that return result, not what you passed

to the add method. Never modify the original entity reference. Avoid keeping any references to the original entity instance.

The recommended pattern is to set a variable with the original entity reference then set its value with the result of the add method. For example:

```
obj = bundle.add(obj)
```

IMPORTANT You must save the **return result** of the add method and use that for any changes. Carefully avoid keeping any references to the original entity instance.

The following example gets the current bundle and moves an entity instance to it

```
uses gw.transaction.Transaction  
  
var bundle = Transaction.getCurrent() // get the current bundle  
  
obj = bundle.add(obj) // move the object to the new bundle and save the result!
```

You can choose to modify the object:

```
obj.Prop1 = "NewValue"
```

You can choose to delete the object, see “Removing Entity Instances from the Database” on page 337:

```
bundle.delete(obj)
```

In most programming contexts, it is extremely critical to not explicitly commit the bundle after your changes. For example, in typical Rules and PCF contexts, it is dangerous and unsupported to explicitly commit the bundle. Let the system perform its default bundle commit behavior. For more information, see “Committing a Bundle Explicitly in Very Rare Cases” on page 337.

Moving a Writable Entity Instance to a New Writable Bundle

As mentioned in “Making an Entity Instance Writable By Adding to a Bundle” on page 334, you can add a read-only entity instance to a writable bundle. You can also add an entity instance to a writable bundle even if the original entity instance is in a writable bundle already. For this action to succeed, the original entity instance must be unmodified. The entity instance must not be newly added, changed, or deleted in its existing bundle.

If you try to add a modified entity instance to a new bundle, Gosu throws the exception `IllegalBundleTransferException`. To avoid this, be careful not to let the entity be modified in more than one bundle.

Be warned that even if the entity instance is unmodified at the time you copy it to the new bundle, there might be problems later. Only one bundle can try to commit a change to the same entity instance based on the same revision of the database row. If an entity instance was modified in more than one bundle and both bundles commit, the second commit fails with a *concurrent data change exception*.

The second commit attempt (the one that fails) might be in code other than your code, such as PCF user interface code that is editing the same object.

Gosu attempts to avoid this problem by preventing adding an already-modified entity to a new bundle. In some cases, user interface code internally *refreshes* an entity instance. For example, switching from view mode to edit mode in the user interface. Refreshing an entity is a special internal process that discards cached versions and gets latest versions from the database. In practice, this process reduces the likelihood of current data change exceptions. This is an internal process only. There is no public API to refresh entity instances from Gosu.

Getting the Bundle of an Existing Entity Instance

If you have a reference to an entity instance in a bundle, you can get a reference to its bundle in its `object.Bundle` property. Use this if you want entity changes to happen together with already-existing changes that you know are in a writable bundle.

For example, you can use this bundle reference and call the add method on to add additional entity instances. See “Making an Entity Instance Writable By Adding to a Bundle” on page 334. If some code commits that bundle, all changes to the database happen together in one database transaction.

Getting an Entity from a Public ID or a Key (Internal ID)

In some code you might not have a reference to an entity instance but you have one of the following:

- **The object public ID** – A public ID is the entity instance’s `PublicID` property. Integration code frequently needs to manipulate objects by public ID.
- **The object key** – A key represents the entity instance’s `Id` property, which is unique for that entity type. Generally speaking, customer code does not manipulate objects by the key. The `Id` property is the underlying primary key for that object and the contents of a typical foreign key reference. In the underlying database row, this column is just a number. In Gosu, the key object contains both the entity type and the ID value.

To load an entity by a key, use the `Bundle` method `loadBean`, which takes a `gw.pl.persistence.core.Key` object. Create a `Key` object with a constructor that takes the entity type and the numeric ID. For example:

```
a = gw.transaction.getCurrent().loadBean(new Key(Address, 123))
```

To load an entity by a public ID, use the query builder APIs. For example:

```
Query.make(Claim).compare(Claim#PublicID, RelOp.Equals, myPublicId).select().AtMostOneRow
```

For more information, see “Query Builder APIs” on page 125. If you use the query API, 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. See “Adding Entity Instances to Bundles” on page 334.

Creating New Entity Instances 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.

Arguments to new operator	Example	Result
<code>no arguments</code>	<code>new Note()</code>	<p>Constructs a new <code>Note</code> entity in the current bundle. Changes related to the current code submit to the database at the same time as any other changes in the current bundle. This approach is the typical approach for most programming contexts, such as rule set code or plugin implementation code. This approach requires that there be a current bundle.</p> <p>In some programming contexts, such as batch processes and WS-I web service implementations, there is no automatic current bundle. In those cases, see “Running Code in an Entirely New Bundle” on page 342.</p>

Arguments to new operator	Example	Result
an entity instance	<code>new Note(myClaim)</code>	Constructs a new Note entity in the same bundle as a given Claim entity instance. Changes to the Claim submit to the database at the same time as the new note, as well as all other changes in the bundle.
a bundle	<code>new Note(myClaim.bundle)</code>	<i>Same as previous table row</i>

For more information about the new operator, see “New Object Expressions” on page 73.

Committing a Bundle Explicitly in Very Rare Cases

Committing a bundle means to send all changes for entity instances in this group of entities to the database. If the commit succeeds, some entity instances may be added, changed, or removed from the database. In typical customer code, do not explicitly commit a bundle. For an overview of related issues, see “When to Use Database Transaction APIs” on page 332.

Generally speaking for data changes, one of the following is true:

- The application has a default bundle management life cycle and commits the bundle at the appropriate time. For example, in rule set execution or typical PCF edit screens. Thus, there is no reason to explicitly commit a bundle.
- There are cases in which you must create an entirely new bundle. See “When to Use Database Transaction APIs” on page 332 and “Running Code in an Entirely New Bundle” on page 342. In such cases, do not explicitly commit the bundle. The `runWithNewBundle` API commits the bundle automatically when your code completes. Thus, there is no reason to explicitly commit a bundle.

WARNING Only commit a bundle if you are sure it is appropriate for that programming context. Otherwise you could cause data integrity problems. For example, in Rule sets or PCF code, it is typically dangerous to commit a bundle explicitly. Contact Customer Support if you have questions.

The most common reason to commit a bundle explicitly is in RPCE web service implementations, which is the older style of web services. In contrast, WS-I web service implementations that need to change data use the `runWithNewBundle` API. The `runWithNewBundle` API automatically commits the bundle for you when your code completes.

If you are sure it is appropriate to commit the bundle, use the method `bundle.commit()`. If the attempt fails, Gosu throws an exception. The entire commit process fails. The database remains unchanged.

Get the current bundle by calling the `getCurrent` static method on the `Transaction` class:

```
uses gw.transaction.Transaction
var bundle = Transaction.getCurrent()
bundle.commit()
```

If you have an entity instance reference, get its bundle in the `entity.bundle` property. It is important to understand that committing a bundle commits everything in the bundle, not just the one instance from which you got the bundle reference. For example:

```
var bundle = myClaim.bundle.commit() // commit *all* entities in the bundle
```

Removing Entity Instances from the Database

Instead of adding or changing entities by adding them to the bundle, mark an entity for removal from the database using the bundle method called `remove`. If the bundle commits successfully, Gosu removes the entity from

the database. If the data model configuration for this entity type declares this entity is retrivable, the entity is not removed and instead is *retired*. See “Overview of Data Entities” on page 157 in the *Configuration Guide*.

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
```

Use the `remove` method carefully. This method dramatically affects data integrity.

The `remove` method deletes or retires data from the database permanently. The `remove` method does not merely remove objects from the bundle. There is no API to remove (evict) an entity instance from the bundle.

WARNING The `remove` method is dangerous. Use it with great care.

Determining What Data Changed in a Bundle

In rule set code, it is sometimes important to identify what data recently changed. 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 contain conditions and rule actions that must detect which if any properties just changed. For example, if a certain object property changed, you might want to validate properties, recalculate related properties, log changes, or send messages to external systems.

To detect data changes, call the object’s `isFieldChanged` method. If it changed, call the entity’s `getOriginalValue` method to get the original value loaded from the database. You must cast 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 differently. The following table describes the behavior based on the property type.

Type of property	Behavior of <code>getOriginalValue</code>	Behavior of <code>isFieldChanged</code>
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, <code>isFieldChanged</code> would return false.
Entity	<p>Guidewire applications represent links to entity subobjects as a foreign key called the <i>internal ID</i>. This foreign key is the <code>entity.Id</code> property on the destination entity. Note that the <code>Id</code> property is different than the <code>PublicID</code> property. If you call <code>getOriginalValue</code> with a foreign key field, you get an internal ID as a Key object. A Key object encapsulates the entity type and the unique ID value for that object.</p> <p>To get a reference to the entity instance with the original ID use the bundle method <code>loadByKey</code>. See "Getting an Entity from a Public ID or a Key (Internal ID)" on page 336.</p> <p>For example:</p> <pre>var k = e.getOriginalValue("Address1") var bundle = Transaction.getCurrent() var addy = bundle.loadByKey(k) as Address</pre> <p>Remember to call the <code>getOriginalValue</code> method on the correct entity instance.</p>	<p>Returns true if and only if the foreign key <code>Id</code> value for the subobject is the same in the original entity. Remember to call the <code>isFieldChanged</code> on the correct entity instance.</p> <p>Keep in mind that checks for a changed entity could return <code>false</code> even if there are changes on properties of subobjects.</p> <p>Be aware that the foreign key is a link to the <code>Id</code> property of the target object. The <code>PublicID</code> property of the target object is not examined.</p>
Array	<p>An array containing the set of array elements that were originally persisted.</p> <p>If any properties changed on those elements, the bundle contains the new values, not the original values. If you need the original properties on those array elements, you must examine each combination of element and properties. In other words, call <code>getOriginalValue</code> on each array element for each property that you are interested in. Also see "Detecting Property Changes on an Entity Instance" on page 340.</p>	<p>Returns true if any elements were added or removed to the array.</p> <p>Additionally, if the array is defined in the data model configuration as an <i>owned array</i>, if any properties on an array element changed, <code>isFieldChanged</code> returns true. This does not apply to non-owned arrays.</p> <p>Note: Because entity array order is not meaningful (do not rely on it), the order is not checked by <code>isFieldChanged</code>. Effectively, in this case <code>isFieldChanged</code> returns the same value as the array method <code>isArrayElementAddedOrRemoved</code>, discussed in "Getting Changes to Entity Arrays in the Current Bundle" on page 340.</p>

Scalar Value Property Example

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`.

Entity Property Example 1

For an entity property 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 instance than the original entity loaded from the database, it is considered changed:

- The expression `claim.isFieldChanged("LossLocation")` returns `true`

- The expression `claim.getOriginalValue("LossLocation")` returns the ID of the original entity instance as a key. Do not assume that the original entity instance is unchanged. It may be loaded and already changed. See “Detecting Property Changes on an Entity Instance” on page 340.

For example:

```
if (claim.isFieldChanged("LossLocation")) {
    // get original entity
    var id = claim.getOriginalValue("LossLocation") as Key
    var originalAddress = claim.Bundle.loadByKey(id)

    // get original property on original entity
    var origAddLine1 = originalAddress.getOriginalValue("AddressLine1")
}
```

In contrast, if the `claim.isFieldChanged("LossLocation")` returns `false`, then the entity foreign key is unchanged but that does not mean necessarily that data in a subobject is unchanged. 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")
    }
}
```

Entity Property Example 2

The following example gets the original value for an entity property and gets a property on that original object

```
var originalPolicyID = account.getOriginalValue("Policy") as Key
var originalPolicy = account.Bundle.loadByKey(originalPolicyID)
var originalPolicyNumber = originalPolicy.getOriginalValue("PolicyNumber") as String
```

Detecting Property Changes on an Entity Instance

From your rule set code, you can use the entity instance read-only property `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...
}
```

From your rule set code, you can get the set of changed properties on an entity instance using its `ChangedFields` property. That property value is a set of type `java.util.Set`.

The following example uses a Gosu block (an in-line function) that iterates across the set of changed properties:

```
if (myAddress.Changed) {
    myAddress.ChangedFields.each( \ e -> print("address has changed property: " + e))
}
```

For more information about blocks, see “What Are Blocks?” on page 231. For information about sets and collections, see “Collections” on page 251.

Getting Changes to Entity Arrays in the Current Bundle

In addition to entity array properties supporting the `isFieldChanged` and `getOriginalValue` methods, you can test for changes to entity array contents in the current bundle. These methods compare the array on the persisted entity instance to the array on the cached version in the bundle.

The following methods work only with properties directly defined on an entity in the data model configuration files as entity arrays. They do not work any other types of properties or methods. For example:

- they do not work with Gosu enhancement properties that return entity arrays
- they do not work with other virtual properties such as those 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 APIs on an entity to get entity array changes

Entity method	Argument	Return type	Returns
getAddedArrayElements	array property name	entity array	Returns a list of array elements that were added. If there are none, returns an empty array.
getChangedArrayElements	array property name	entity array	Returns a list of array elements that were changed. If there are none, returns an empty array.
getRemovedArrayElements	array property name	entity array	Returns a list of array elements that removed (marked for deletion if the bundle commits). If there are none, returns an empty array.
isArrayElementChanged	array property name	boolean	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.
isArrayElementAddedOrRemoved	array property name	boolean	Returns true if and only if any array elements were added or removed.

Getting Add, Changed, or Deleted Entities In a Bundle

A bundle's set of entity instances might include some combination of the following types of entities. You can get this set of changes using properties on the bundle object. The following table lists the APIs you can use for this purpose.

Type of entities to return	Bundle API	Description
New entity instances	<code>bundle.InsertedBeans</code>	Entity instances in this bundle that are entirely new objects (never committed to the database) appear in this set. This bundle property returns an iterator that iterates across the set of entity instances.
	<code>bundle.getInsertedBeansOfType(t)</code>	Same as <code>InsertedBeans</code> , but as a method that allows you to filter only those of a specific entity type.
New and updated entity instances	<code>bundle.InsertedAndUpdatedBeans</code>	Entity instances in this bundle that are entirely new objects (never committed to the database) or changed appear in this set. This bundle property returns an iterator that iterates across the set of entity instances.
	<code>bundle.getInsertedAndUpdatedBeansOfType(t)</code>	Same as <code>InsertedAndUpdatedBeans</code> , but as a method that allows you to filter only those of a specific entity type.
Changed entity instances	<code>bundle.ChangedBeans</code>	Entity instances in this bundle that had one or more properties change appear in this set. This bundle property returns an iterator that iterates across the set of entity instances.
	<code>bundle.getChangedBeansOfType(t)</code>	Same as <code>ChangedBeans</code> , but as a method that allows you to filter only those of a specific entity type.

Type of entities to return	Bundle API	Description
Removed entities	bundle.RemovedBeans	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 <i>retired</i> . This bundle property returns an iterator that iterates across the set of entity instances.
	bundle.getRemovedBeansOfType(t)	Same as RemovedBeans, but as a method that allows you to filter only those of a specific entity type.
Unmodified entity instances in the current bundle	n/a	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.

Running Code in an Entirely New Bundle

Typical Gosu code interacts with database transactions by using the current bundle set up for rules execution, plugin code, or PCF user interface code. In such cases, ClaimCenter automatically manage the low level parts of database transactions. Only in rare cases your code must create or commit a bundle explicitly, such as batch processes, WS-I web service implementation classes, and actions triggered from workflows.

In rare cases, you can run Gosu code in an entirely new bundle. In other words, you sometimes must create a different transaction. You might need a new transaction because there is no current bundle or because you need changes in a different transaction from the current transaction.

Create your own bundle only in the following cases:

- If your code modifies entities to commit independent of the **success** of the surrounding code. For example, some workflow asynchronous actions or unusual PCF user interface code might need to create a new bundle.
- If your code represents a separate asynchronous actions from surrounding code
- 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 an in-line 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 231.

To create a new bundle, call the static method `runWithNewBundle` on the `gw.transaction.Transaction` class. As a method argument, pass a Gosu block that takes one argument, which is a bundle (Bundle). Gosu creates an entirely new bundle 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 entity instances to the bundle as appropriate using the `bundle.add(entity)` method. Remember to save the return result of the add method. See “Adding Entity Instances to Bundles” on page 334.

Gosu runs this code synchronously (immediately). 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, throw an exception from the Gosu block. Gosu does not commit the bundle. The bundle is discarded.

The following example demonstrates how to create a new bundle to run code that creates a new entity instance and modifies some fields. The current transaction is an argument to the block.

If you are adding entity instances, in typical code you do not need to use the bundle explicitly. Gosu sets the current bundle inside the block to this new block automatically. You can use the no-argument version of the `new` operator for entity types, which creates the object the current bundle:

```
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
}
```

If you perform database queries, you can use the bundle reference to add entity instances to the new writable bundle. See “Adding Entity Instances to Bundles” on page 334.

WARNING Only commit a bundle if you are sure it is appropriate for that programming context. Otherwise you could cause data integrity problems. For example, in Rule sets or PCF code, it is typically dangerous to commit a bundle explicitly (including with this API). Contact Customer Support if you have questions.

Using Variable Capturing in Your Block

The following example method from a web service implementation demonstrates how to create a new bundle to call a utility function that does the core part of the work.

```
...
public function myAction(customerID : String) : void {
    Transaction.runWithNewBundle( \ bundle -> MyClass.doSomething(customerID, bundle))
}
...
```

Although this example looks simple, the block is not entirely self-contained. It uses the method argument `customerID` for the `myAction` method even though `customerID` is not declared in the scope of the block. The ability for the block to use variables from its creation context is called *variable capturing*. For more information about this feature, see “Variable Scope and Capturing Variables In Blocks” on page 234.

In some cases, you might need to return information back from your block. You can do this using more advanced use of variable capturing. The following example demonstrates variable capturing by creating a local variable in the outer scope (`myAction`) 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 myAction(customerID : String) : boolean {
    var res : boolean

    // the local variable "res" and the parameter "customerID" is captured by the block.
    // Both variables are shared between the calling function and the block
    Transaction.runWithNewBundle( \ bundle -> { res = MyClass.doSomething(customerID, bundle) })

    // return the value that was set in the block using the shared (captured) variable
    return res
}
```

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 as 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 instance 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` entity type or type list.

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 And Database Commits

Generally speaking, if your code throws an exception, the current database transaction does not commit. This applies to rule sets, PCF code, and other context with a current bundle.

Throw an exception from your code to get this behavior where appropriate. If you catch any exceptions in your code and you want to ensure the current bundle is not committed in the current action, rethrow the exception.

Bundles and Published Web Services

There are two types of web services that you can publish from ClaimCenter:

- WS-I web services (the newer style)
- RPCE web services (the legacy style of web services)

For WS-I web services, all web service implementation classes must explicitly manage all bundles, both for reading and writing entity data. However, WS-I web services do not support entity instances as arguments or return types, so the interaction between web service implementation classes and bundles is straightforward.

For RPCE web services, it is more complex because RPCE web services support entity instances as arguments and return types. There is important built-in behavior for handling entity data in RPCE web services that you must understand to implement these web services. For more information, see “Committing Entity Data in Your Web Service” on page 101 in the *Integration Guide*.

WARNING RPCE web services are deprecated. Convert existing RPCE code to WS-I web services.

Entity Cache Versioning, Locking, and Entity Refreshing

The application caches entity data for faster access. All APIs discussed in this topic that refer to *original* entity instances 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 database row from the database long before the recent application change.

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.

Entity Instance Versioning and the Entity Touch API

ClaimCenter protects entity instances from concurrent access through a version property that exists on all *versionable entities*. Almost all entity types in the system are versionable. If you load a versionable entity instance into a bundle, the application loads the version number and stores it with the data.

If a bundle commits, ClaimCenter checks this version number property in each entity instance with the latest version in the database. ClaimCenter confirms that the cached entity instance is up to date. If the version numbers match, the commit can proceed.

If the version numbers do not match, the entire commit attempt fails for the entire bundle. In other words, if the recent change for an object relies on an out-of-date database row, it is unsafe to commit this recent change. Gosu throws an exception during the database commit.

During the final commit, the version number is increased by one from its previous value. In typical code, you need to add entity instances to a bundle and then modify them. ClaimCenter automatically increments the version number.

In rare cases, it may be desirable to force the version number of an entity to increment even if there is no known change to the entity yet.

For example, suppose your code reads values from four entity instances (A, B, C, and D) that have a tight conceptual relationship. Perhaps your code reads the values and changes most or all of the objects in one database transaction. In some case, perhaps the algorithm makes change to only B, C, and D. There remains one object (A) for which you did not explicitly change properties. Therefore, A is not in the same *bundle* as B, C and D. The database row for A does not update nor does the version number increment. For some circumstances, they may not be a problem.

However, suppose the three objects you change are related to the *current* properties on A. The default behavior may be undesirable compared to updating A, B, C, and D together:

- You might want to protect against other threads on the current server (including the user interface) from making concurrent changes on A that make the other changes make no sense.
- You might want to protect against other servers in the ClaimCenter cluster from making concurrent changes on A that make the other changes make no sense.
- You might want the last modified time of A to match the last modified time of B, C, and D.
- You might want to force pre-update rules to run for object A.

To force ClaimCenter to increment the entity version number (`obj.BeanVersion`), update the modified time, and force pre-update rules to run, call the `touch` method on the entity instance. The method takes no arguments:

```
obj.touch()
```

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 internally refreshes bundle entity data with the latest version from the database in cases that seem 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 Bundle Commit Steps

If ClaimCenter commits a bundle to save data to the database, the following steps occur during bundle commit:

1. The application reserves a connection from the connection pool.
2. Internally, the application saves the state of the bundle. The application uses this snapshot to revert to the current state of data if something fails during writing to the database.
3. The application runs pre-update rule sets.
4. The application refreshes all entity instances that already existed but are unchanged in the bundle. This ensures that validation rules, which run next, get the newest versions of those entity instances.
5. The application triggers validation rule sets.
6. The application sets properties that exist on editable and versionable entities: `updateTime`, `createTime`, and `user`.
7. The application increments the version number on each modified entity instance.
8. Any entity instances that are new (not yet in the database) have a temporary assigned internal ID. An internal ID is the `entity.Id` property. At this step, the application assigns the actual permanent internal ID for the entity instances. The application fixes any foreign key references to refer to the new internal ID number rather than the temporary assigned ID.
9. The application computes the set of new, changed, and removed entity instances in the bundle.
10. The application determines the set of messaging events that are raised by changes in the bundle.
11. For each messaging destination that is registered to listen for messaging events, the application triggers Event Fired rule sets once for each messaging destination. For example, `ENTITYNAMEAdded`, `ENTITYNAMEChanged`, or `ENTITYNAMERemoved` events. If more than one destination listens for the same event, the Event Fired rule set executes once for each messaging destination. See “Messaging and Events” on page 299 in the *Integration Guide* for much more information on this topic.
12. The application writes all changed entities to the database connection. It is during this step that the application checks for concurrent data change exceptions. See “Entity Instance Versioning and the Entity Touch API” on page 345.
13. The application attempts to commit the database connection. This either completely succeeds or fails.
14. The application updates the global entity cache. The global entity cache speeds up entity data access by caching recently used data in memory on each server in an application cluster.
15. After updating the cache, the current server signals other servers in the cluster about the change. This signal tells the other servers to remove data from the cache for entity instances that just updated or deleted. Entity instances in remote entity cache immediately show the data as requiring an update, but does not immediately reload from the database.
Any local existing references to the old data on the other servers are out of date and will never commit to the database. The entity data is safe from accidental changes of this type. See “Entity Instance Versioning and the Entity Touch API” on page 345.

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” on page 347
- “Using Template Files” on page 349
- “Template Export Formats” on page 353

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 349.

The simplest way to use 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 347. 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 347, 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 350.
- **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 351.
- **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 352.

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 351 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 351 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 `${}` 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 351 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 198.

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.

Type System

Gosu provides several ways to gather information about an object or other type. Use this information for debugging or to change program behavior based on information gathered at run time.

This topic includes:

- “The Type of All Types” on page 355
- “Basic Type Coercion” on page 356
- “Basic Type Checking” on page 357
- “Using Reflection” on page 360
- “Compound Types” on page 367
- “Type Loaders” on page 367

The Type of All Types

The Type data type is a meta-type. It is the type of Gosu types. If you get the type of something using the `typeof` keyword, the type of the result is Type. In some cases in APIs, you will see the interface type `IType`, which also refers to a 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 355.

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 357.

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
<code>(5 * 4) as Array</code>	n/a	Type mismatch or possible invalid operator. <code>java.lang.Object[]</code> is not compatible with <code>java.lang.Double</code>
<code>(5 * 4) as Boolean</code>	Boolean	true
<code>(5 * 4) as DateTime</code>	DateTime	1969-12-31 (default value)
<code>(5 * 4) as Key</code>	n/a	Type mismatch or possible invalid operator. <code>com.guidewire.commons.entity.Key</code> is not compatible with <code>java.lang.Double</code>
<code>(5 * 4) as String</code>	String	20
<code>(5 * 4) as Type</code>	n/a	Type mismatch or possible invalid operator. <code>MetaType:java.lang.Object</code> is not compatible with <code>java.lang.Double</code>

Why Use Coercion?

Gosu requires all variables to have types at compile time. 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 358.

For related information, see “Basic Type Checking” on page 357 and “Using Reflection” on page 360.

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
<code>42 typeis Number</code>	<code>true</code>
<code>"auto" typeis String</code>	<code>true</code>
<code>person typeis Person</code>	<code>true</code>
<code>person typeis Company</code>	<code>false</code>

Similarly, you can use the `typeof object` operator to test against a specific type.

Typeof Syntax

```
typeof expression
```

Typeof Examples

Expression	Result
<code>typeof 42</code>	<code>Number</code>
<code>typeof "auto"</code>	<code>String</code>
<code>typeof (4 + 5)</code>	<code>Number</code>

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 360.

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 360.

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 92.
- **switch** statements. For more information, see “Switch() Statements” on page 95. 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 80.

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 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 is 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 runs 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 an object is, you can use *reflection* to learn about the type or perform actions on it. Reflection means using type introspection to query or modify objects at runtime. For example, instead of simply calling an object method, get a list of methods from its type, or call a method by name specified by a `String` run time value. You can get metadata, properties and functions of a type at run time.

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()  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 357 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 198.
IsArray	If the type is an array, returns true.
IsFinal	If the type is final, returns true. See “Modifiers” on page 198.
IsGeneric	If the type is generic, returns true. See “Gosu Generics” on page 239.
IsInterface	If the type is an interface, returns true. See “Interfaces” on page 211.
IsParameterized	If the type is parameterized, returns true. See “Gosu Generics” on page 239.
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 Values” on page 47.

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.

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 `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.

In rare cases, you might want to do reflection on the backing class (underlying implementation Java class) as opposed to the Gosu type. For example, suppose you need to inspect each field in the original Java class, rather than the Gosu type that references it. You can get the backing class by casting the type to `IHasJavaClass`, for example:

```
// get the Java backing class
var javaClass = (theType as IHasJavaClass).BackingClass

// With that class, you can iterate across the native Java fields at run time
for( field in javaClass.Fields ) {
    print(field)
}
```

The `IHasJavaClass` interface is implemented only by types that have a Java backing class.

For a related topic, see “Calling Java from Gosu” on page 119.

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.getByName( 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.

Feature Literals

Gosu feature literals provide a way to statically refer to a type's features such as methods and properties. You can use feature literals to implement some reflection techniques more safely at compile time than in some programming languages. Gosu feature literals are validated at compile time, preventing many common errors with reflection techniques. Refer to the feature of an type or object using the # operator after the type or object, followed by the feature name. The syntax is similar to feature syntax in the Javadoc @link syntax.

You can use feature literals in APIs such as:

- Mapping layers, where you are mapping between properties of two types
- Data-binding layers
- Type-safe object paths for a query layer

Consider the following Gosu class:

```
class Employee {  
    var _boss : Employee as Boss  
    var _name : String as Name  
    var _age : int as Age  
  
    function update( name : String, age : int ) {  
        _name = name  
        _age = age  
    }  
  
    function greeting() {  
        print("hello world")  
    }  
}
```

Given this class, you can refer to its features by using the # operator:

```
// get property feature literal  
var nameProp = Employee#Name  
  
// get method feature literal  
var updateFunc = Employee#update(String, int)
```

These variables contain feature literal references. Using these feature references, you can access the underlying property or method information. Alternatively, use feature references to directly invoke a method or get/set a property.

You can define or use Gosu APIs that use feature literals as arguments or return values.

Get or Set a Property

To get or set a property, call the get or set method on the feature literal and pass the object instance of the appropriate type as a method argument.

Using the feature literal code from the previous example:

```
var anEmp = new Employee() { :Name = "Joe", :Age = 32 }  
  
// get property using normal syntax  
print( anEmp.Name ) // prints "Joe"  
  
// GET a property using a feature literal  
var nameProp = Employee#Name  
print(nameProp.get(anEmp))  
  
// SET a property using a feature literal  
nameProp.set( anEmp, "Ed" )  
print( anEmp.Name ) // now prints "Ed"
```

Call a Method

Using the feature literal code from the previous example, you can call a method:

```
var m = Employee#greeting()  
var obj = new Employee()
```

```
// call the method and pass the object with the method feature literal on the left of the period
m.invoke(obj)
```

You can also call a method with arguments:

```
var m = Employee#update(String, int)

var obj = new Employee()

// call method with arguments after the object instance reference
m.invoke(obj, "John", 25)

// test results...
print(obj.Name + " and age " + obj.Age)
```

Bind a Property to an Instance Rather Than a Type

You can bind a feature literal to a specific object instance. This technique is the equivalent of requesting the property by name for a particular instance of the object using the compile time type.

Using the feature literal code from the previous example, this code sets a property:

```
var anEmp = new Employee() { :Name = "Joe", :Age = 32 }

// get the property feature literal for a specific instance
var namePropForAnEmp = anEmp#Name

// set a property value using that feature literal
namePropForAnEmp.set( "Ed" )

print( anEmp.Name ) // prints "Ed"
```

You do not need to pass the instance into the `set` method because the code already bound the property reference to the `anEmp` variable.

Bind Argument Values in Method References

You also can bind argument values in method references:

```
var anEmp = new Employee() { :Name = "Joe", :Age = 32 }
var updateFuncForAnEmp = anEmp#update( "Ed", 34 )

print( anEmp.Name ) // prints "Joe", code did not yet invoke the function reference

// call the method with specific arguments bound in the feature literal
updateFuncForAnEmp.invoke()

print( anEmp.Name ) // prints "Ed" now
```

Using this technique, you can refer to a method invocation with a particular set of arguments. Note that the second line does not invoke the `update` function. Instead, it gets a reference that you can use to evaluate the function at a later time.

Chaining Feature Literals

Feature literals support typesafe chaining, so you could write this code:

```
var bossesNameRef = anEmp#Boss#Name
```

The feature literal refers to the name of the boss of the object in the `anEmp` variable.

Convert Method References to Gosu Blocks

You can convert method references to blocks with the `toBlock` method:

```
var aBlock = anEmp#update( "Ed", 34 ).toBlock()
```

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 239.

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 215.

Type Loaders

The Gosu type system has an open type system. An important part of this is that Gosu supports custom type loaders. A type loader dynamically injects types into the language and attaches potentially complex dynamic

behaviors to working with the type. A custom type loader adds types to the type system and optionally runs custom code every time any code accesses properties or call methods on them.

There are several built-in type loaders:

- **Gosu XML/XSD type loader.** This type loader supports the native Gosu APIs for XML. For more information, see “Gosu and XML” on page 267.
- **Gosu SOAP/WSDL type loader.** This type loader supports the native Gosu APIs for the web services SOAP protocol. This works through a Gosu type loaders that reads web service WSDL files and lets you interact with the external service through a natural syntax and type-safe coding. For more information, see “Calling WS-I Web Services from Gosu” on page 71 in the *Integration Guide*.
- **Property file type loader.** This type loader finds property files in the hierarchy of files on the disk along with your Gosu class files. Gosu creates types in the appropriate package (by the property file location) for each property. You can access the properties directly in Gosu in a type-safe manner. For more information, see “Properties Files” on page 393.

You do not need to do anything special to install or enable these type loaders. Gosu includes these type loaders automatically for all Gosu code.

Concurrency

This topic describes Gosu APIs that protect shared data from access from multiple threads.

This topic includes:

- “Overview of Thread Safety and Concurrency” on page 369
- “Request and Session Scoped Variables” on page 371
- “Concurrent Lazy Variables” on page 372
- “Concurrent Cache” on page 373
- “Concurrency with Monitor Locks and Reentrant Objects” on page 374

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 a 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:

- **Request and session variables.** The classes `RequestVar` and `SessionVar` in the package `gw.api.web` synchronize and protect access to shared data. These APIs return a variable that is stored in either the request or the session. See “Request and Session Scoped Variables” on page 371.
- **Lazy concurrent variables.** The `LockingLazyVar` 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 `LockingLazyVar` class uses the Java concurrency libraries, access to the lazy variable is thread-safe. The `LockingLazyVar` class wraps the double-checked locking pattern in a typesafe holder. See “Concurrent Lazy Variables” on page 372
- **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. See “Concurrent Cache” on page 373.

WARNING Caches are difficult to implement and use. Caches can cause subtle problems. Use caches only as a last result for performance. If you use a cache, it is best to request multiple people on your team carefully review 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. See “Concurrency with Monitor Locks and Reentrant Objects” on page 374.

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 Database Transactions” on page 331.

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 Database Transactions” on page 331.

Request and Session Scoped Variables

To create a variable that can safely be accessed in the request or session scope, use the following two classes in the `gw.api.web` package:

- `RequestVar` – Instantiate this class to create a request-scoped variable stored in the web request
- `SessionVar` – Instantiate this class to create a session-scoped variable stored in the web session

These classes create a variable with a well-defined lifecycle and attachment point of either the request or session.

Use the `set` method to set the variable value. Use the `get` method to get the variable value.

For both objects, from Gosu there is a `RequestAvailable` property. Check the value of this `boolean` property to see if there is a request (for `RequestVar`) or a session (for `SessionVar`) available in this programming context.

For example, if the code was called from a non-web unit test or a batch process, these properties return `false`. If `RequestAvailable` returns `false`, it is unsafe to call either `get` or `set` methods on them. From Java, these properties appear as the `isRequestAvailable` method on each object.

WARNING Rule sets, plugin code, and other Gosu code can be invoked through various code contexts. You may not encounter contexts other than the user interface until late in system testing. When getting values `RequestVar` and `SessionVar`, always use `RequestAvailable` before trying to access the variable. Always handle the variable not being available or if it returns a `null` value.

The recommended pattern for concurrent use is to store the instance of the variable as a class variable declared with the `static` keyword. By using the `static` keyword on the class, varied programming contexts can access the same session variable or request variable. Use the generics syntax `SessionVar<TYPE>` when you define the variable. For example, to store a `String` object, define the type as `SessionVar<String>`. For more information about Gosu generics, see “Gosu Generics” on page 239.

For example:

```
class MyClass {  
    static var _varStoredInSession = new SessionVar<String>()  
  
    static property get MySessionVar() : String {  
        if ( _varStoredInSession.RequestAvailable ) {  
            return _varStoredInSession.get()  
        } else {  
            return null // ENSURE THE CALLER CHECKS FOR NULL  
        }  
    }  
  
    static property set MySessionVar( value: String ) {  
        if ( _varStoredInSession.RequestAvailable ) {  
            _varStoredInSession.set( value )  
        }  
        else {  
            // decide what to do if the request is not available  
        }  
    }  
}
```

In any other part of the application, you can write code that set this property:

```
MyClass.MySessionVar = "hello"
```

In another part of the application, you can write code that gets this property:

```
print( MyClass.MySessionVar )
```

Note that the example explicitly checks the `RequestAvailable` property. In real code, you must decide what you want to do if `RequestAvailable` returns `false` to support your batch processes for example.

It is strongly recommended to use `RequestVar` and `SessionVar` rather than the Java thread local API `java.lang.ThreadLocal<TYPE>`. Because application servers pool their threads, using `ThreadLocal` increases risks of data that stays around forever as a memory leak, since the thread you attached it to never dies. Also, pooled threads can preserve stale data from a previous request, since the server thread pool reuses the thread for future requests. If you ever use a `ThreadLocal`, it is critical to be very careful to clean up all code carefully with a `try/finally` block. Alternatively, convert that code to use `RequestVar` and `SessionVar`.

Concurrent Lazy Variables

In addition to using the Java native concurrency classes, Gosu includes utility classes that add additional concurrency functionality. The `LockingLazyVar` class implements what some people call a *lazy variable*. This means Gosu constructs it only the first time some code uses it. Because the `LockingLazyVar` class uses the Java concurrency libraries, access to the lazy variable is thread-safe. The `LockingLazyVar` class wraps the double-checked locking pattern in a typesafe holder.

In Gosu, you will see the `make` method signature

`LockingLazyVar.make(gw.util.concurrent.LockingLazyVar.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 `LockingLazyVar` value. An example is easier to understand than the method signature. The following example passes a block as an argument to `LockingLazyVar.make(...)`. That block creates a new `ArrayList` parameterized to the `String` class:

```
var _lazy = LockingLazyVar.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 : LockingLazyVar<List<String>> = LockingLazyVar.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 = LockingLazyVar.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.

Optional Non-Locking Lazy Variables

Gosu also provides a non-locking variant of `LockingLazyVar` called `LocklessLazyVar`. Use `LocklessLazyVar` if you do not need the code to be thread-safe. Because it does not lock, it performs faster and has less chance of server deadlock. However, it is unsafe in any contexts that have potential concurrent access and thus require thread safety.

Concurrent Cache

A similar class to the `LockingLazyVar` 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 functionUpperCaseQuickly( 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 retVal : Object
        _lock.with( \-> {
            retVal = _someVar.someMethod()
        })
        return retVal
    }
```

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 syntax works with lock objects because Gosu considers this objects *reentrant*.

For important information `using` clause syntax, see “Object Lifecycle Management (`using` Clauses)” on page 104. For example, that topic describes how Gosu handles multiple resources initialized or locked in a single clause

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 package: `ReentrantLock`, `ReadWriteLock`, `Condition`.
- You cast the object to the Gosu interface `IMonitorLock`. You can cast **any** arbitrary object to `IMonitorLock`. It is useful to cast Java monitor locks to this Gosu interface. For more information on this concept, refer to: [http://en.wikipedia.org/wiki/Monitor_\(synchronization\)](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.

For more information about `using` clauses, see “Object Lifecycle Management (using Clauses)” on page 104.

Note: For more information about concurrency and related APIs in Java, see:

<http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html>

Gosu Command Line Shell

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.

Gosu Command Line Tool 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

Unpacking and Installing the Gosu Command Line 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
<code>/bin/gosu.cmd</code>	The Windows tool that invokes Gosu
<code>/src/gw/.../*.gs</code>	Core Gosu classes
<code>/src/gw/.../*.gsx</code>	Core Gosu enhancements
<code>/lib/*.jar</code>	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.

Command Line Tool Options

The following table lists the tool command line options:

Task	Options	Example
Run a Gosu program. Include the <code>.gsp</code> file extension when specifying the file name.	<code>filename</code>	<code>gosu myprogram.gsp</code>
Default behavior of command line tool with no options is same as interactive shell (<code>-i</code>) option		<code>gosu</code>
Enter interactive shell. Each line you type runs as a Gosu statement. Any results print to the standard output. To exit, type the <code>exit</code> or <code>quit</code> command. For details, see "Gosu Interactive Shell" on page 382. Also see the standard input option (just a hyphen), discussed later in this table.	<code>-i</code> <code>-interactive</code>	<code>gosu -i</code>
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 382	<code>-</code> <i>(just the hyphen character)</i>	From DOS command prompt: <pre>echo print(new DateTime()) gosu -</pre>
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 <code>></code> and <code><</code> , precede them with a caret (<code>^</code>) symbol.	<code>-e expression</code> <code>-eval expression</code>	<code>gosu -e "new DateTime()"</code> <code>gosu -e """a"""+"""b"""</code>
Add additional paths to the search path for Java classes or Gosu classes. Separate paths with semicolons. If you are running a <code>.gsp</code> file, it is often easier to instead use the <code>classpath</code> command within the <code>.gsp</code> file rather than this option. For related information, see class loading information in "Setting the Class Path to Call Other Gosu or Java Classes" on page 386 and "Advanced Class Loading Registry" on page 381.	<code>-classpath path</code>	<code>gosu -classpath C:\gosu\projects\libs</code>
Print help for this tool.	<code>-h</code> <code>-help</code>	<code>gosu -h</code>

Writing a Simple Gosu Command Line Program

The following instructions describe running a basic Gosu program after you install the gosu command line tool. These instructions apply only to the command line shell for Gosu. If you are using the Gosu plugin for the IntelliJ IDEA IDE, to get command line Gosu you must download the full distribution at:

<http://gosu-lang.org/downloads.html>

To run a program

1. Create a file called myprogram.gsp containing only the following line:

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

2. Open a command prompt.

3. Change your working directory to the directory with your program.

4. Type the following command :

```
gosu myprogram.gsp
```

If you have not yet added the gosu executable to your system path, instead type the full path to the gosu executable. For more information, see “Unpacking and Installing the Gosu Command Line Shell” on page 377

5. The tool runs the program.

The program outputs the following:

```
Hello World
```

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 `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 in this example (`Name`) requires an argument value to follow the option. This is true of all non-boolean property types.

However, the `boolean` property in this example (`Hidden`) does not require an argument value, and this type is special for this reason. If a property is declared with 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 declared with modifiers `public`, `static`, and `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
<code>@Required()</code>	This command line tool will not parse unless this property is included
<code>@DefaultValue(String)</code>	The default string value of this property.
<code>@ShortName(String)</code> <code>@LongName(String)</code>	By default, Gosu also exposes the property name automatically with the single hyphen option. If the property uses mixed case (<i>camel case</i>) after the first letter, Gosu converts each capital letter to an underscore and the lowercase letter. The short and long name of this option. <ul style="list-style-type: none"> • Gosu uses the short name when used with a hyphen and one letter argument. • Gosu uses the long name when used with a double hyphen and one-or-more letter argument For example, suppose you use the following annotations with the property called <code>DataBaseChoice</code> . <code>@ShortName("d")</code> <code>@LongName("database")</code>
	You can use either of the following arguments to the command line tool: <ul style="list-style-type: none"> • <code>-data_base_choice</code> • <code>-d</code> • <code>--database</code>

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>..</entry>
        <entry>..</entry>
    </classpath>
```

```
</serialization>
```

To add paths to the class path, add more `<entry>` elements containing class paths.

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
<code>quit</code>	Quit the interactive shell.
<code>exit</code>	Quit the interactive shell
<code>ls</code>	Show a list of all defined variables
<code>rm VARNAME</code>	remove a variable from interactive shell memory
<code>clear</code>	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” and then redisplays the current line you are typing:

```
each( block( java.lang.Object ):void )
elementAt( int )
equals( java.lang.Object )
eachWithIndex( block( java.lang.Object, int ):void )
ensureCapacity( int )
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 read 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.

Gosu Programs

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.

For more information about command line use, see “Gosu Command Line Shell” on page 377

The following instructions describe running a basic Gosu program after you install the gosu command line tool. These instructions apply only to the command line shell for Gosu. If you are using the Gosu plugin for the IntelliJ IDEA IDE, to get command line Gosu you must download the full distribution at:

<http://gosu-lang.org/downloads.html>

The Structure of a Gosu Program

A simple Gosu program is one or more lines that contain Gosu statements. There are several important other elements of a Gosu program:

- “Metaline as First Line” on page 385
- “Functions in a Gosu Program” on page 386
- “Setting the Class Path to Call Other Gosu or Java Classes” on page 386

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 in a Gosu Program

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
```

Setting the Class Path to Call Other Gosu or Java Classes

You can call out to any Java or Gosu class as needed. However, you cannot define Gosu classes directly inside your Gosu program file.

To tell Gosu where to load additional classes, do either of the following:

- Use the `classpath` argument on the command line tool. See “Command Line Arguments” on page 379.
- Add a `classpath` statement to the top of your Gosu program.

The `classpath` statement in a Gosu program improves upon the Java approach, which is to invoke a full and long `classpath` argument option when running the main class.

To add to the class path for a program from within the program, simply add `classpath` statements before all other statements in the program. If you use a metaline (see “Metaline as First Line” on page 385), `classpath` statements appear after the metaline.

The simple version of the `classpath` statement is simply a relative path in quote signs, for example:

```
classpath "src"
```

The first character of the path is important to determine the type of path:

- If the path starts with the forward slash (“/”) character, Gosu assumes it is an *absolute path*.
- If the path starts with a character other than a forward slash (“/”) character, Gosu assumes it is 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.

You can include multiple paths in the same string literal using a comma character as a separator.

Typically you place Java classes, Gosu classes, or libraries in 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 file for the class `mypackage.MyClass` to the location:

```
C:\gosu\myprograms\test1\src\mypackage\ MyClass.class
```

Copy your library files to locations such as:

```
C:\gosu\myprograms\test1\lib\mylibrary.jar
```

For this example, add classpath values with the following statement:

```
classpath "src,lib"
```

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.withStdErrorHandler(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 383.

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” on page 389
- “Creating Fingerprints” on page 390
- “Extending Fingerprints” on page 391

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.

Gosu 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 \rightarrow 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 bytes 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
```


Properties Files

Gosu includes automatic support for reading properties files in the Java properties format.

This topic includes:

- “Reading Properties Files” on page 393

Reading Properties Files

Gosu includes automatic support for reading properties files in the Java properties format. Gosu accomplishes this with a custom type loader that adds types in the type system for any file with the `.properties` file extension in the class hierarchy. The location of the file within the class hierarchy defines the package (namespace) for created types. Gosu creates a type that matches the name of the properties file without the file extension. The following procedure describes in detail how to use this feature.

To read a properties file from Gosu

1. Find your root of your class hierarchy.

If your Gosu code is in a Gosu program (a `.gsp` file), you can add a root directory to your class path using the `classpath` statement. See “Setting the Class Path to Call Other Gosu or Java Classes” on page 386.

2. Decide where in your package hierarchy that you want to reference your properties file. For example, suppose the root of your class hierarchy is the path `/MyProject/gsrc`. If you want your properties file to be in the package `com.mycompany.config` and the properties file to be called `MyProps.properties`, create a new file at the path:

`/MyProject/gsrc/com/mycompany/config/MyProps.properties`

3. In that file, add the following content:

```
# the hash character as first char means the line is a comment
! the exclamation mark character as first char means the line is a comment

website = http://gosu-lang.org/
language = English

# The backslash below tells the application to continue reading
# the value onto the next line.
message = Welcome to \
```

```
Gosu!  
  
# Unicode support  
tab : \u0009  
  
# multiple levels of hierarchy for the key  
gosu.example.properties.Key1 = Value1
```

A few things to notice:

- The message property definition uses multiple lines, using the backslash to continue reading from the next line.
- The tab property definition uses Unicode syntax with \u followed by four hexadecimal digits for the Unicode code point.
- The last property in the file uses multiple levels of hierarchy

4. To test this code from another Gosu class, use the following code:

```
uses com.mycompany.config.*  
  
print("accessing properties...")  
print("")  
  
print(" message: ${MyProps.message}")  
print(" website: ${MyProps.website}")  
print(" gosu.example.properties.Key1: ${MyProps.gosu.example.properties.Key1}")  
print(" unicode support (tab char): before${MyProps.tab}after")
```

Run this code to print the following:

```
accessing properties...  
  
message: Welcome to Gosu!  
website: http://gosu-lang.org/  
gosu.example.properties.Key1: Value1  
unicode support (tab char): before      after
```

To test this code with a Gosu program instead of a Gosu class, create a Gosu program called `PropsTest.gsp` one level higher than the root of your class hierarchy. Add a `classpath` statement to add the root of the class hierarchy to the class path. See “Setting the Class Path to Call Other Gosu or Java Classes” on page 386.

Limitations of the Properties File Type Loader

The properties file type loader does not support key values with spaces, or any other characters that would be illegal in a Gosu property name. Gosu omits any such properties.

For example, the following Java property file includes a key with a name that includes embedded spaces using the backslash character before each space character,

```
# Add spaces to the key  
key\ with\ spaces = This is the value that could be looked up with the key "key with spaces".
```

Although it is a legal Java property, Gosu does not provide programmatic access to it.

Coding Style

This topic lists some recommended coding practices for the Gosu language. 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” on page 395

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 69

Capitalization Conventions

The following table lists conventions for capitalization of various language elements:

Language element	Standard capitalization	Example
Gosu keywords	Always specify Gosu keywords correctly as they are declared, typically lowercase. Java keywords are case-sensitive.	if
type names, including class names	uppercase first character	DateUtil Claim
local variable names	lowercase first character	myClaim
property names	uppercase first character	CarColor
method names	lowercase first character	printReport
property names	uppercase first character	Name
package names	lowercase all letters in packages and subpackages	com.mycompany.*

However, because Gosu is case-sensitive, you must access existing types items exactly as they are declared, including the correct capitalization. Capitalization in the *middle* of a word is also important.

Use the Gosu editor Code Completion feature to enter the names of types and properties correctly. This ensures standard capitalization.

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 Gosu supports public variables for compatibility with other languages, the standard Gosu style is to use public properties backed by private variables rather than public variables. You can do this easily in Gosu on the same line as the variable definition using the `as` keyword followed by the **property name**.

In other words, in your new Gosu classes that define class variables, use this variable declaration syntax:

```
private var _firstName : String as FirstName
```

This declares a private variable called `_firstname`, which Gosu exposes as a public property called `FirstName`.

Do not do this:

```
public var FirstName : String // do not do this. Public variable scope is not Gosu standard style
```

For more information about defining properties, see “Properties” on page 193.

IMPORTANT For Gosu classes data fields, the standard Gosu style is to use public properties backed by private variables rather than public variables. Do not use public variables in new Gosu classes. See “Properties” on page 193 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 358.