Guidewire PolicyCenter™

Gosu Reference Guide

Release 10.0.0

GUIDEWIRE
Adapt and succeed™

Product Name: Guidewire PolicyCenter
Product Release: 10.0.0
Document Name: Gosu Reference Guide
Document Revision: 25-September-2018

Contents

Part 2
Advanced Gosu features

Part 3
Gosu programs

Part 4
Testing

Part 5
Gosu style

About PolicyCenter documentation

The following table lists the documents in PolicyCenter documentation:

Document	Purpose
InsuranceSuite Guide	If you are new to Guidewire InsuranceSuite applications, read the *InsuranceSuite Guide* for information on the architecture of Guidewire InsuranceSuite and application integrations. The intended readers are everyone who works with Guidewire applications.
Application Guide	If you are new to PolicyCenter or want to understand a feature, read the *Application Guide*. This guide describes features from a business perspective and provides links to other books as needed. The intended readers are everyone who works with PolicyCenter.
Database Upgrade Guide	Describes the overall PolicyCenter upgrade process, and describes how to upgrade your PolicyCenter database from a previous major version. The intended readers are system administrators and implementation engineers who must merge base application changes into existing PolicyCenter application extensions and integrations.
Configuration Upgrade Guide	Describes the overall PolicyCenter upgrade process, and describes how to upgrade your PolicyCenter configuration from a previous major version. The intended readers are system administrators and implementation engineers who must merge base application changes into existing PolicyCenter application extensions and integrations. The *Configuration Upgrade Guide* is published with the Upgrade Tools and is available from the Guidewire Community.
New and Changed Guide	Describes new features and changes from prior PolicyCenter 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.
Installation Guide	Describes how to install PolicyCenter. The intended readers are everyone who installs the application for development or for production.
System Administration Guide	Describes how to manage a PolicyCenter system. The intended readers are system administrators responsible for managing security, backups, logging, importing user data, or application monitoring.
Configuration Guide	The primary reference for configuring initial implementation, data model extensions, and user interface (PCF) files for PolicyCenter. The intended readers are all IT staff and configuration engineers.
PCF Reference Guide	Describes PolicyCenter PCF widgets and attributes. The intended readers are configuration engineers.
Data Dictionary	Describes the PolicyCenter data model, including configuration extensions. The dictionary can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers.
Security Dictionary	Describes all security permissions, roles, and the relationships among them. The dictionary can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers.
Globalization Guide	Describes how to configure PolicyCenter for a global environment. Covers globalization topics such as global regions, languages, date and number formats, names, currencies, addresses, and phone numbers. The intended readers are configuration engineers who localize PolicyCenter.
Rules Guide	Describes business rule methodology and the rule sets in Guidewire Studio for PolicyCenter. The intended readers are business analysts who define business processes, as well as programmers who write business rules in Gosu.
Contact Management Guide	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 PolicyCenter implementation engineers and ContactManager administrators.

Document	Purpose
Best Practices Guide	A reference of recommended design patterns for data model extensions, user interface, business rules, and Gosu programming. The intended readers are configuration engineers.
Integration Guide	Describes the integration architecture, concepts, and procedures for integrating PolicyCenter 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.
Java API Reference	Javadoc-style reference of PolicyCenter Java plugin interfaces, entity fields, and other utility classes. The intended readers are system architects and integration programmers.
Gosu Reference Guide	Describes the Gosu programming language. The intended readers are anyone who uses the Gosu language, including for rules and PCF configuration.
Gosu API Reference	Javadoc-style reference of PolicyCenter Gosu classes and properties. The reference can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers, system architects, and integration programmers.
Glossary	Defines industry terminology and technical terms in Guidewire documentation. The intended readers are everyone who works with Guidewire applications.
Product Model Guide	Describes the PolicyCenter product model. The intended readers are business analysts and implementation engineers who use PolicyCenter or Product Designer. To customize the product model, see the *Product Designer Guide*.
Product Designer Guide	Describes how to use Product Designer to configure lines of business. The intended readers are business analysts and implementation engineers who customize the product model and design new lines of business.

Conventions in this document

Text style	Meaning	Examples
italic	Indicates a term that is being defined, added emphasis, and book titles. In monospace text, italics indicate a variable to be replaced.	A *destination* sends messages to an external system. Navigate to the PolicyCenter installation directory by running the following command: `cd installDir`

bold	Highlights important sections of code in examples.	`for (i=0, i<someArray.length(), i++) {`
		`  newArray[i] = someArray[i].getName()`
		`}`

narrow bold	The name of a user interface element, such as a button name, a menu item name, or a tab name.	Click **Submit**.
`monospace`	Code examples, computer output, class and method names, URLs, parameter names, string literals, and other objects that might appear in programming code.	The `getName` method of the `IDoStuff` API returns the name of the object.
`monospace italic`	Variable placeholder text within code examples, command examples, file paths, and URLs.	Run the `startServer` *`server_name`* command. Navigate to `http://`*`server_name`*`/index.html`.

Support

For assistance, visit the Guidewire Community.

Guidewire customers

`https://community.guidewire.com`

Guidewire partners

`https://partner.guidewire.com`

part 1

Gosu language basics

chapter 1

Gosu introduction

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

Welcome to Gosu

Welcome to the Gosu language. Gosu is a general-purpose programming language built on top of the Java Virtual Machine (JVM). Because Gosu uses the JVM, Gosu includes the following features:

- Java compatible, so you can use Java types, extend Java types, and implement Java interfaces
- Object-orientation
- Easy to learn, especially for programmers familiar with Java
- Imperative paradigm

Gosu includes the following additional features:

- *Static typing*, which helps you find errors at compile time.
- *Type inference*, which simplifies your code and preserves 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.
- XML/XSD support.
- Web service (SOAP) support.

Large companies around the world use Gosu every day in production systems for critical applications.

Basic Gosu

The following Gosu program uses the built-in `print` function to write the text `"Hello World"` to the default output stream or console:

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

Gosu uses the Java type `java.lang.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 simplify common text substitution coding patterns.

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

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

Although this example does not specify types for the variables, Gosu is statically typed. All variables have a compile-time type that Gosu enforces at compile time, even though this example has no explicit type declaration. Gosu automatically assigns the type `int` to these variables. Gosu infers the type `int` from the expressions on the right side of the equal signs on lines that declare the variables. Inferring the type of an item from the expression that initializes the item is called *type inference*.

Type inference simplifies Gosu code compared to other statically typed programming languages. Type inference 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 pattern looks like:

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

This line is equivalent to the following Java code:

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

The Gosu version is more readable and concise than the Java version.

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.

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.

See also

- “Generics in Gosu” on page 41
- “Type inference” on page 39
- “More about the Gosu type system” on page 38
- “Gosu statement terminators” on page 42

Overview of control flow

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

if statements

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`. Alternatively, you can use the symbolic versions from Java (`&&`, `||`, and `!`).

for loop

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

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

This syntax accepts arrays or any `Iterable` object. Although the syntax does not specify a type for `ad`, 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`, Gosu skips the `for` statement entirely, and generates no error. In contrast, Java throws a null pointer exception if the iterable object is `null`.

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

```
for (ad in addressList index i) {
  print(ad.ID + " has index " + i)
}
```

Intervals

Gosu has native support for intervals. An *interval* is a sequence of values of the same type between a given pair of endpoint values. For example, the set of integers beginning with 0 and ending with 10 is an integer interval. A *closed interval* contains the starting and ending values, in this example, including 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. The Gosu syntax for this closed interval is `0..10`. A typical use for intervals is in writing concise `for` loops:

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

An *open interval* excludes the value at one or both ends of the interval. The Gosu syntax `1|..|10` defines an interval that is open on both sides, and contains the values from 2 through 9.

Intervals do not need to represent numbers. Intervals can be of types including numbers, dates, or other abstractions such as names. Gosu accepts the shorthand syntax of the two periods, (`..`) for intervals of dates and common number types. You can also add custom interval types that support iterable comparable sequences. If your interval type implements the required interfaces, you can use intervals of that type 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)
```

In practice, using intervals makes most use of this pattern unnecessary. If necessary, you can use a Gosu `while` loop to duplicate this pattern.

To use intervals with `for` loops, the interval must be iterable. Custom non-iterable intervals are useful mainly for math and theoretical work. For example, a non-iterable interval could represent non-countable values such as the infinite set of real numbers between two other real numbers.

switch statement

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 continues to the next case in the series. In the previous example, if you remove the `break` statements, the code prints both `"b"` and `"c"`. Java has the same requirement. Some other languages do not require the `break` statement to prevent continuing to the next case.

See also

- "Intervals" on page 131

Overview of blocks

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

Blocks are useful as method parameters, in which the method's implementation generalizes a task or algorithm but callers provide code to implement the details of the task. For example, Gosu adds many useful methods to Java collections classes that take a block as a parameter. That block could return an expression, such as a condition against which to test each item, or could represent an action to perform on each item.

For example, the following Gosu code makes a list of strings, sorts the list by `String` length, and 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) })
```

Block shortcut for one-method interfaces

If an anonymous inner class implements an interface that has exactly one method, you can use a Gosu block to implement the interface. The Gosu block is an alternative to using an explicit anonymous class. You can use a Gosu block for interfaces that are defined in either Gosu or Java. For example:

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

See also

- "Blocks" on page 175
- "Gosu block shortcut for anonymous classes implementing an interface" on page 168

Define a block

A block defines an in-line function. For example, a block named `square` multiplies a value by itself and returns the result.

Procedure

1. Start with the `\` character.
2. Optionally add a list of arguments as *`name : type`* pairs.
3. Add the `->` characters, which mark the beginning of the block's body.
4. Finally, add either a statement list surrounded by braces: `{` and `}`, or a Gosu expression.

Example

The following block multiplies a number with itself to square the number:

```
var square = \ x : Integer-> x * x // No need for braces for expressions, but must use for statements
var myResult = square(10)          // Call the block
```

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

Overview of enhancements

Gosu provides a feature called *enhancements*, which supports adding functions and properties to other types. Enhancements are 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 and interfaces, such as `java.util.List`, that improve power and readability of collections code. For example, the following code takes a list of `String` objects, and uses a block to sort the list by the length of each `String`. The code then uses another block in the iteration across the result list to print each item:

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

The `sortBy` and `each` methods in this example are Gosu enhancement methods on the `List` interface. Both methods return the result list, which makes them useful for chaining methods in series.

See also

- "Enhancements" on page 201
- "Blocks" on page 175

Overview of collections

Gosu provides several features for usage of collections like lists and maps. Gosu directly uses the built-in Java collection classes like `java.util.ArrayList` and `java.util.Hashmap`. This usage makes Gosu straightforward to use for interaction with pre-existing Java classes and libraries.

In addition, Gosu adds the following features:

- Shorthand syntax for creating lists and maps that is natural 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 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.

```
// 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. Alternatively, for readability, you can put each step on a separate lines. The following example declares some data, then uses a block to search for a subset of the items, and then sorts the results.

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

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

See also

- “Overview of blocks” on page 22

Overview of access to Java types

Gosu provides full access to Java types. Gosu is built on top of the Java language and runs within the Java Virtual Machine (JVM). Gosu loads all Java types, so you have full direct access to Java types, such as classes, libraries, and primitive (non-object) types. You can use your favorite Java classes or libraries directly from Gosu with the same syntax as for native Gosu objects.

The most common Java object types retain their fully qualified name but are always in scope, so you do not need to fully qualify their names in typical code. Examples of classes that are always in scope are `Object`, `String`, and `Boolean`. If your code has ambiguity because of similarly named types also in scope, you must fully qualify these types, such as `java.lang.String`. Other Java types are available to Gosu code, but must be fully qualified, for example `java.io.DataInputStream`.

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

```
var list = new ArrayList<String>()
list.add("Hello Java, from Gosu")
```

Gosu includes transformations on Java types that make your Gosu code clearer, such as turning getters and setters into Gosu properties.

Access to Java types from Gosu includes:

- Instantiation of Java classes with the `new` keyword
- Implementing Java interfaces
- Manipulating Java objects as native Gosu objects
- Calling object methods on instantiated objects
- Exposing object methods that look like getters and setters as properties
- Calling static methods on Java types
- Providing built-in extensions and improvements of many common Java types by using Gosu enhancements
- Support for your own extensions of Java types and interfaces
- Support for Java primitive types

See also

- “Packages always in scope” on page 116
- “Overview of enhancements” on page 23
- “Calling Java from Gosu” on page 139

Overview of 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 of this class. A *package* defines a namespace. 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 specifies 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 example shows a simple class with one constructor that requires a `String` argument:

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

You can optionally specify that your class implements one or more interfaces.

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, four more lines contain the object variable name, which is repeated information.

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

```
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 uses more lines but is more readable:

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

Unlike in 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 dense hierarchical structure, omitting the type name can make your Gosu code more readable.

See also

- “Overview of interfaces” on page 29
- “Creating and instantiating classes” on page 148
- “Using the operator new in object expressions” on page 86
- “Importing types and package namespaces” on page 114

Functions

Declare a function by using the `function` keyword. Gosu uses the term *method* for a function that is part of another type. 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, which can be 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. You call such functions directly in your code. A rare globally available function for any Gosu code is `print`. Call that function with a `String` argument to write data to the system console or other default output stream. For example, the following line prints text to the console:

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

Gosu supports access control modifiers (`public`, `private`, `internal`, and `protected`). These modifiers have the same meaning as in Java. For example, if you mark a method `public`, any other code can call that method.

Gosu also supports static methods, which are methods on the type rather than on object instances.

If the return type for a method is not `void`, all possible code paths in the method must contain a `return` statement and return a value. The set of all paths includes all outcomes of conditional execution, such as `if` and `switch` statements. This requirement is identical to the analogous requirement in Java. The Gosu editor notifies you at compile time if your code violates this requirement.

See also

- “Access modifiers” on page 157
- “Static members” on page 28
- “Gosu functions” on page 111

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 // Variables are private by default
```

One difference between Gosu and some languages, including Java, is full support in Gosu for properties. A *property* provides a dynamic getter method and a dynamic setter method for its value. To set or get properties from an object, use natural syntax. Type the period (.) character followed by the property name just as for an object variable:

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

Internally, Gosu calls the property getter and setter methods. In addition, Gosu has a special null-safety behavior with pure property paths, which have the form `obj.Property1.Property2.Property3`.

Define a property accessor function, which is a property getter, by using the declaration `property get` instead of `function`. Define a setter function by using function declaration `property set` instead of `function`. These property functions dynamically get or set the property, depending on whether the definition is `property get` or `property set`. Properties can be read/write, 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`* in the definition for a

variable. This syntax separates the internal implementation of the variable from how you expose the property to other code:

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

This syntax is a shortcut for creating a `property get` function and a `property set` function for each variable. This syntax is the standard and recommended Gosu style for designing public properties. Do not expose actual class variables as public, although for compatibility with Java, Gosu does support exposing these variables.
The following code defines a simple Gosu class:

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

class Person {

  var _name : String as Name    // Exposes the _name field as a readable and writable 'Name' property
  var _id : String              // Variables 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
    _name = ""
  }

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

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

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

With this class, you can 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!
```

See also

- “Property accessor paths are null safe” on page 28

Java getter and setter methods become properties in Gosu

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 method as a setter for the 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 method as a property set function for the property named `Name` of type `String`.
- If the method name starts with `get` and takes no arguments and returns a value, Gosu exposes this method 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 method as a `property get` function for the property named `Name` of type `String`.
- If the method name starts with `is`, takes no arguments, and returns a Boolean value, the rules are similar to the rules for `get`. Gosu exposes this method as a getter for the property. The property name matches the original

method but without the prefix `is`. For example, suppose a Java method signature is `isVisible()`. Gosu exposes this method as a `property get` function for a Boolean property named `Visible`.

If the Java code has 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. With this property, you can use straightforward code such as:

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

See also

- "Java get, set, and is methods convert to Gosu properties" on page 141

Property accessor paths are null safe

One notable difference between Gosu and some other languages is that property accessor paths in Gosu are tolerant of unexpected `null` values. This feature 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 tend 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.

Gosu provides 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)"
```

WARNING In Studio, Gosu code in the Gosu Scratchpad has different compiler behavior than any other part of Studio. If you run code in Gosu Scratchpad directly in Studio without being connected to a running server, the code compiles and runs locally in Studio with different behavior. Most notably, the behavior of null safety in the period operator is different. In nearly all other contexts in Studio, the period operator is null safe. However, in Gosu Scratchpad in Studio without a connection to a running server, Gosu is not null-safe. Code that might not throw an exception in a Gosu class might throw an exception in Gosu Scratchpad. If Studio is connected to a running server, Studio sends the Gosu code to the server. The PolicyCenter server uses the standard null-safe behavior for the period operator.

See also

- "Null safety for properties and other operators" on page 39

Static members

Gosu supports static members on a type. Static members include variables, functions, property declarations, and static inner classes on a type. A *static* member exists only once, on the type, not on instances of the type.

To declare a type as static for a new Gosu class, use the `static` keyword, as in Java.

The syntax for using a static member is to use the period (.) character followed by the property name or method call after a type reference, which is the type name. For example, the following code accesses a static property and calls a static method:

```
var myVar = MyClass.MY_CONST // Get a static property value
MyClass.staticMethodName()   // Call a static method
```

Alternatively, to use the static members without qualifying them with the class name, you use the syntax of the `uses` statement that imports static members. For example, the following code is equivalent to the previous lines:

```
uses MyClass#*
...
var myVar = MY_CONST // Get a static property value
staticMethodName()   // Call a static method
```

See also

- "Modifiers" on page 157
- "Importing static members" on page 116

Overview of interfaces

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

```
package example

interface ILoadable {
  function load()
}
```

A class can implement the interface with the `implements` keyword followed by a comma-delimited list of interfaces. A class that implements an interface contains all methods in the interface, as shown in the following code:

```
package example

class LoadableThing implements ILoadable {

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

See also

- "Interfaces" on page 171

List and array expansion operator *.

Gosu provides an operator for array expansion and list expansion that expands and flattens complex object graphs. This expansion can be useful and powerful, enabling you to extract 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
```

Using the expansion operator on a list retrieves a property from every item in the list and returns all instances of that property in a new list. The expansion operator 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 previous expression is a list containing the same number of items as the original list. However, each item is the length (the `String.length` property) of the corresponding name `String` object.

Gosu uses Gosu generics, an advanced type feature, to infer the type of the list as the appropriate parameterized type. Similarly, Gosu infers the type of the result array if you call the operator on an array.

See also

- "Generics in Gosu" on page 41
- "List expansion (*.)" on page 187

Overview of comparisons

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

- The operators `>`, `<`, `>=`, and `<=` operators work with all objects that implement the `Comparable` interface, not just with 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 that if the value on either side of the operator is `null`, Gosu does not throw a null pointer exception.

  Note: In contrast, in the Java language, the `==` operator evaluates to `true` if and only if both operands have exactly the same reference value. The Java expression evaluates to `true` if both terms refer to the same object in memory. This behavior provides the expected result for primitive types like integers. For reference types, the references are not usually what you want to compare. Instead, to compare value equality, Java code typically uses *`object`*`.equals()`, rather than the `==` operator.
- In some cases, you do need to compare identity references, to determine whether two objects reference the same in-memory object. Gosu provides a special equality operator called `===` (three equal signs) to compare object equality. This operator compares whether both references point to the same in-memory object. The following examples illustrate some differences between `==` and `===` operators:

Expression	Output	Description

`var x = 1 + 2`
`var str = x as String`
`print(str == "3")`
	`true`	These two comparison arguments reference the same value but different objects. Using the double-equals operator returns `true`.

`var x = 1 + 2`
`var str = x as String`
`print(str === "3")`
	`false`	These two comparison arguments reference the same value but different objects. Using the triple-equals operator returns `false`.

See also

- "Property accessor paths are null safe" on page 28

Case sensitivity

Gosu language itself is case sensitive. Write all Gosu as case-sensitive code matching the declaration of the language element.

See also

- "Gosu case sensitivity and capitalization" on page 42

Compound assignment statements

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

- `++` – The increment-by-one operator, which is supported only after the variable name
- `+=` – The add-and-assign operator, which is 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.

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

```
i++
```

These operators always form statements, not expressions. For example, the following Gosu is valid:

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

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, so `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. In Gosu, these operators do not form an expression, so you cannot use increment or decrement in `while` declarations, `if` declarations, and `for` declarations.

See also

- “Variable assignment” on page 103

Delegating interface implementation with composition

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

```
interface IClipboardPart {
  function load()
}

class ClipboardPart implements IClipboardPart {
  override function load() {
    print("Gosu is loaded!")
  }
}

class MyWindow implements IClipboardPart {
  delegate _clipboardPart represents IClipboardPart

  construct() {
    _clipboardPart = new ClipboardPart(this)
    this.load()
```

```
  }
}
```

In this example, the class definition for `MyWindow` uses the `delegate` keyword to delegate implementation of the `IClipboardPart` interface. To complete the delegation, the constructor for the `MyWindow` class creates an object of the `ClipboardPart` class. This object is the delegate. It has the delegate's name, `_clipboardPart`. When a class delegates the implementation of an interface, it must construct the delegate it declares. Upon doing so, the class fulfills indirectly its implementation duty.

The class that defines a delegate object must implement the interface that the delegate represents. In the example code, the `ClipboardPart` class defines the delegate object, `_clipboardPart`. This definition implements the sole method that the `IClipboardPart` interface declares, `load()`.

When instantiating the `MyWindow` class, the class constructor in the example code calls the constructor for the class that defines the delegate object, `_clipboardPart`. Upon its construction, `_clipboardPart` has access to all of the methods that its defining class, `ClipboardPart`, implements, including the `load()` method. Upon constructing its delegate, the `MyWindow` class has access to the `load()` method as well.

The class constructor in the example code then calls the `load()` method. This method in turn calls the `print()` method. The `print()` method then writes the sentence "Gosu is loaded!" to the system console or default output stream.

You can use a delegate to represent (provide methods for) multiple interfaces for the enclosing class. In the `delegate` statement, 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 by using a special kind of type called a compound type.

See also

- "Composition" on page 249

Overview of 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. The requirement to support concurrent access from multiple threads is called *concurrency*. Code that can safely get or set concurrently accessed data 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 provides straightforward access to these classes with `using` clauses that also properly handle cleanup if exceptions occur. Additionally, Gosu provides a straightforward way to create custom Gosu objects that support reentrant object handling, as shown in the following example. The following Gosu code that uses the `using` keyword shows the compact readable syntax for using Java-defined reentrant locks.

  ```
  // 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` provide synchronization and protect access to shared data. These APIs return `Map` objects that you can use to safely get and put data by using different scope semantics.
- **Lazy concurrent variables** – The `LazyVar` class in `gw.util.concurrent` implements what is known as a lazy variable. Gosu does not construct a *lazy variable* until the first time any code uses the variable. For example, the following code is part of a class definition that defines the object instance. For example, Gosu does not run the following Gosu block that creates an `ArrayList` until the first usage of the variable:

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

- **Concurrent cache** – The `Cache` class in `gw.util.concurrent` declares a cache of values that you can look up quickly and in a thread-safe way. A `Cache` object provides a concurrent cache similar to a Least Recently Used (LRU) cache. To use the cache, call the `get` method with an argument of the input value, which is the key. If the value is in the cache, the `get` method returns the value from the cache. If the value is not cached, Gosu calls the block and calculates the value from the key, and then caches the result. For example:

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

See also

- "Concurrency" on page 417

Overview of 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 in Gosu, but if the exception is not a `RuntimeException`, Gosu wraps the exception in a `RuntimeException`.

Catching exceptions

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

```
try {
  user.enter(bar)
} catch (e : Exception) {
  print("Failed to enter the bar!")
} finally {
  // Clean-up code here...
}
```

To catch only specific exceptions, specify a subclass such as `IOException` instead of `Exception`. 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`, which has a `Message` property that contains the `String` data. Your error-handling code can use the `Message` property for logging or displaying messages to users.

You could rewrite the previous `throw` code example as the concise code:

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

Overview of annotations

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

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)
```

See also

- “Annotations” on page 209

Overview of templates

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

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

If you print this variable, Gosu displays:

```
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 line 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.

See also

- “Templates” on page 301

Overview of XML and XSD support

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

Gosu can parse XML by using an existing XML Schema Definition (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`:

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

The following Gosu code uses XSD-based types to manipulate XML objects:

```
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
}
```

The example to follow uses the XSD file, `Credentials.xsd`. In particular, the code relies upon the following portion of the XSD file:

```
<xs:complexType name= "CredentialsType">
  <xs:sequence>
    <xs:element name="username" type="xs:string" minOccurs="1" maxOccurs= "1"/>
    <xs:element name="password" type="xs:string" minOccurs= "1" maxOccurs= "1"/>
  </xs:sequence>
  <xs:attribute name="key" type="xs:string"/>
</xs:complexType>
```

```
<xs:element name="CredentialsElem" type="tns:CredentialsType"/>

<xs:complexType name="CredentialsArrayType">
  <xs:sequence>
    <xs:element ref="tns:CredentialsElem" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="CredentialsArray" type="tns:CredentialsArrayType"/>
```

In this example, the following Gosu code creates a first element defined in the XSD file. This first element is of type `CredentialsArray`. Next, the code creates a second element defined in the same XSD file. This second element is of type `CredentialsElem`. The code then adds the second element as a child to the first element:

```
uses gw.xsd.credentials.*

var e = new CredentialsArray()

// Create a new child element that is legal in the XSD, and add it as a child
var c1 = new CredentialsElem()
c1.setUsername("Harry Truman")
c1.setPassword("password")
e.addChild(c1)
```

See also

- "Working with XML in Gosu" on page 253

Overview of web service support for consuming WSDL

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

The following example uses a hypothetical 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()
```

See also

- *Integration Guide*

Gosu character set

Because Gosu runs within a JVM, Gosu shares the same 16-bit Unicode character set as Java. By supporting this character set, your Gosu code can represent virtually any character in any human language.

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, such as Java classes.

You can run Gosu programs (`.gsp` files) directly from the command line. You cannot run a Gosu class file or other types of file directly. If you want to call a Gosu class or other type of file, make a Gosu program that uses those other types.

In Java, you define a `main` method in a class and tell Java which main class to run. This method calls 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 a Gosu class or Java class.

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

- Use the `Class-Path` attribute in the manifest of a JAR file.
- Use the `-classpath` option on the command-prompt tool.

You can combine these two options for greater flexibility and to support very long class paths.

Using the Class-Path attribute in the manifest of a Java library file

You can add a `Class-Path` attribute line in the `MANIFEST.MF` file, which is the manifest file in any Java library file. The value of the `Class-Path` attribute has an arbitrary length, so you can use this attribute if you need a class path that is longer than a command prompt supports.

The `Class-Path` attribute can contain one or more URLs for JAR file names. To specify multiple JAR files in the class path, use a space character to separate the file names. To specify an absolute path to a local file use the `file:/` prefix.

The following lines show a manifest file that contains a `Class-Path` attribute. The class path contains a library in the same directory as the library that contains the manifest, a library in a subdirectory, and a library at an absolute path.

```
Manifest-Version: 1.0
Class-Path: MyUtils.jar classfiles/MyClasses.jar file:/C:/javalib/AppHelpers.jar
Created-By: 1.8.0_92 (Oracle Corporation)
```

Using the -classpath option on the command-prompt tool

To use other Gosu classes, Java classes, or Java libraries, you perform the following steps.

1. Create a package-style hierarchy for your class files on your disk. For example, if the root of your files is `PolicyCenter/MyProject/`, put the class files for the Gosu class `com.example.MyClass` at the location `PolicyCenter/MyProject/com/example/MyClass.gs`.
2. At the command prompt, tell Gosu where to find your other Gosu classes and Java classes by adding the `-classpath` option to the `gosu` command.

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 the location:

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

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

```
test.gsp -classpath "src;lib"
```

See also

- “Programs” on page 437
- “Command-prompt tool and options” on page 432

Guidewire internal methods

Some code packages contain Guidewire internal methods that are reserved for Guidewire use only. Gosu code written to configure PolicyCenter must never call an internal method for any reason. Future releases of PolicyCenter can change or remove an internal method without notification.

The following packages contain Guidewire internal methods.

- All packages in `com.guidewire.*`
- Any package whose name or location includes the word `internal`

Gosu configuration code can safely call methods defined in any `gw.*` package (except for those packages whose name or location includes the word `internal`).

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, rather than a dynamically typed language. In a *statically typed* language, 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 violation as a compile error. If your code violates type constraints at run time, for example, by making an invalid object type coercion, Gosu throws an exception.

Static typing in Gosu provides several benefits:

- Compile-time error prevention
- Intelligent code completion

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. Programmers familiar with the Java implementation of generics quickly become comfortable with Gosu generics, although there are minor differences.

See also

- "Intelligent code completion and other Gosu editor tools" on page 39
- "Type system" on page 403
- "Variable type declaration" on page 102
- "Generics" on page 221

Compile-time error prevention

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

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

- Calls to this function must take as parameters the correct number of parameters having the appropriate types.
- Within the code for the function, code must always treat an 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, run-time errors can occur.
- If code that calls this function 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
}
```

If you write code that calls this method and passes an integer instead of a `Person`, the code fails with a type mismatch compiler error. The compiler recognizes that 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 calls any methods on the object, Gosu also ensures that the method name that you are calling actually exists on the type.

Within the Gosu editor, violations of these rules become compilation errors. By finding this large class of problems at compile time, you can avoid unexpected run-time failures.

Type inference

Gosu supports type inference, in which Gosu can often infer the type of an item 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.

The Gosu syntax for explicit declarations of the type of the variable during declaration is:

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

For typical code, Gosu coding style is to omit the type and use type inference to determine 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 use a class called `Vehicle`, which has 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 and easily. The editor also uses the type system to 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 allowed at that location in the code.

Similar to the use of the period character, 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 to enter an incorrect type, Gosu displays an error message immediately so that you can fix your errors at compile time.

Refactoring

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

See also

- *Configuration Guide*

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. Only expressions separated by period (`.`) characters that access a series of instance variables or properties support null safety, such as the following 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 tend to simplify real-world code. Often, a `null` expression result has the same meaning whether the final property access is `null` or earlier parts of the path are `null`. For such cases in Gosu, do not check for the `null` value at every level of the path. This conciseness makes your Gosu code easier to read and understand.

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

```
house.Walls.Windows
```

In some languages, you must be aware that if `house` is `null` or `house.Walls` is `null`, your code throws a null-pointer exception. The following common coding pattern avoids the exception:

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

// Check earlier parts of the path for null to avoid a null-pointer exceptions (NPE)
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`. The following Gosu code is equivalent to the previous example and a null-pointer exception never occurs:

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

By default, method calls are not null-safe. If the right side of a period character is a method call, Gosu throws a null-pointer exception (NPE) if the value on the left side of the period is `null`.

For example:

```
house.myaction()
```

If `house` is `null`, Gosu throws an NPE. Gosu assumes that method calls might have side effects, so Gosu cannot skip the method call and return `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 the period: `?.`

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

A *null-safe method call* does not throw an exception if the left side of the period character evaluates to `null`. Gosu just returns `null` from that expression. Using the `?.` operator calls the method with `null` safety:

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

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

Null-safe versions of other operators

Gosu provides null-safe versions of other common operators:

- The null-safe default operator (`?:`). You use this operator to specify an alternative value if the value to the left of the operator evaluates to `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. The expression 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 by 2, or return null if cost is null
```

Design code for null safety

Use null-safe operators where appropriate. These operators make code more concise and simplify the handling of edge cases.

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

See also

- “Handling null values in expressions” on page 96
- “Properties” on page 151

Generics in Gosu

Generics abstract the behavior of a type to support working with multiple types of objects. Generics are particularly useful for implementing collections such as lists and 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 >). This syntax that specifies the type of a collection or another item of unknown type is called *parameterizing a generic type*. For example:

```
uses java.util.*

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

In this example, `ArrayList<Date>` is an array list of date objects. The `Map<String, Date>` is a map that maps `String` to `Date`.

The Gosu generics implementation is compatible with the Java 1.5 generics implementation, and adds more 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 support reified generics. *Reified generics* preserve generic type information at run time. In complex cases, you can 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 that you do not need to see the generics syntax, which tends to be verbose. For example, consider the following Gosu:

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

  Although this code does not specify a type, the `strlist` variable is statically typed. Gosu detects the types of objects that you use to initialize a variable and uses type inference to determine that `strlist` has the type `java.util.ArrayList<java.lang.String>`. This generics syntax means "a list of `String` objects."

See also

- "Generics" on page 221

Gosu primitive types

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

Every Gosu primitive type (other than the special value `null`) has an equivalent object type that the Java language defines. 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 primitives in typical code. However, they are slightly different types, just as in Java, and on rare occasion these differences are important, especially with respect to the ability to represent `null`.

Boxed primitives can represent `null` but unboxed primitives cannot represent `null`. Converting a boxed primitive to an unboxed primitive has a special behavior if the variable or expression has the value `null` at run time. For Boolean values, the `null` becomes `false`. For numeric values, the `null` becomes 0 or the closest equivalent, such as the `char` value 0.

Gosu case sensitivity and capitalization

Gosu is case-sensitive. For example, if a type is declared as `MyClass`, you cannot use the type as `myClass` or `myclass`.

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.

See also

- "General coding guidelines" on page 465

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 that Gosu code looks cleaner.

For typical code, omit semicolons because they are unnecessary in almost all cases. Standard Gosu style uses semicolons between multiple Gosu statements when they are all on one line. For example, a short Gosu block definition uses this style. However, even in those cases, 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 : Integer, y : Integer -> { print("I added!"); return x + y; }
```

Valid but not recommended

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

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

// NO - 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 In almost all cases, omit semicolon characters in Gosu. However, standard Gosu style uses semicolons between multiple Gosu statements on one line, such as in short Gosu block definitions.

Invalid statements

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

See also

- “Blocks” on page 175

Gosu comments

Add comments to your Gosu code to explain its programming logic. Gosu supports several styles of comments.

Block comments	Use block comments to describe the purpose of classes and methods. A block comment begins with a slash/asterisk (`/*`) and ends with an asterisk/slash (`*/`).
	`/*`
	`* This is a block comment.`
	`* Use block comments at the beginnings of files and before`
	`* classes and functions.`
	`*/`
	Use proper capitalization and punctuation in block comments.

Single-line comments	Use single-line comments to describe one or more statements within a function or method. A single-line comment begins with double slashes (`//`) as the first non-whitespace characters on the line.
	`if (condition) {`
	`  // Handle the condition`

```
    ...
    return true
  }
```

If you cannot fit your comment on a single line, use a sequence of single-line statements.

Trailing comments	Use trailing comments to briefly describe single lines of code. A trailing comment begins with double slashes (//) as the first non-whitespace characters after the code that the comment describes.
	`if (a == 2) {`
	`  return true  // Desired value of 'a'`
	`}`
	`else {`
	`  return false // Unwanted value of 'a'`
	`}`
	If you place two or more trailing comments on lines of code in a block, indent them all to a common alignment

You can use some styles of comments to temporarily disable lines or blocks of code during development and testing. This technique is known as commenting out code.

Commenting out a single line of code	Use a single-line comment delimiter (//) to comment out a line of code.
	`if (x = 3) {`
	`  // y = z   This line is commented out for testing.`
	`  y = z + 1  This line will excecute for testing.`
	`}`

Commenting out a block of code	Use a pair of block comment delimiters (/*, */) to comment out a block of code, even if the block you want to comment out contains block comments.
	`/*`
	`  /*`
	`   * The function returns the sum of its parts.`
	`   */`
	`  public function myMethod(int A, int B) {`
	`  return A + B`
	`  }`
	`*/`
	The preceding function cannot be called, because it is commented out. Gosu permits the nested block comment within the commented out block of code.

See also

- *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 some of the keywords in the list in the current Gosu grammar. Gosu does not prevent the use of some keywords in contexts outside their Gosu language meaning. Nonetheless, take care not to use any of the following words other than in their Gosu grammar usage. Using a keyword in a non-standard manner creates code that is difficult to understand or fails to compile.

• abstract	• iterator
• and	• length
• annotation	• long
• as	• NaN

• assert	• new
• block	• not
• boolean	• null
• break	• or
• byte	• out
• case	• outer
• catch	• override
• char	• package
• class	• print
• classpath	• private
• construct	• property
• contains	• protected
• continue	• public
• default	• readonly
• delegate	• represents
• do	• return
• double	• set
• else	• short
• enhancement	• startswith
• enum	• static
• eval	• statictypeof
• extends	• structure
• false	• super
• final	• switch
• finally	• this
• float	• throw
• for	• transient
• foreach	• true
• function	• try
• get	• typeas
• hide	• typeis
• if	• typeloader
• implements	• typeof
• in	• uses
• index	• using
• Infinity	• var
• int	• void
• interface	• where

- `internal`
- `while`

Legal variable names

Gosu variable names must conform to the following rules.

The first character of the name can be:

- Any letter
- `$`
- `_`

Subsequent characters can be:

- Any letter
- Any digit
- `$`
- `_`

Gosu generated documentation (Gosudoc)

You use the `gwb` tool to generate the Gosu documentation from the command line:

```
gwb gosudoc
```

This command generates the documentation at `PolicyCenter/build/gosudoc/index.html`.

The formatting of generated Gosu documentation has Javadoc style. This Gosudoc contains the output of the Gosu type system and provides better links between Gosu class definitions than the Gosu API Reference available within Studio.

Code coverage support

Code coverage tools analyze the degree of testing of programming code. For Gosu code in Studio, PolicyCenter 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:

```
PolicyCenter/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 tables briefly summarize notable differences between Gosu and Java, with particular attention to requirements for converting existing Java code to Gosu. If the Gosu column contains multiple code examples, these examples show the multiple ways in which Gosu can represent the Java code in the Java column. If the Required Change? column contains “Required”, you must make this change to convert existing Java code to Gosu. If this column contains “Optional”, that item is either an optional feature, a new feature, or Gosu optionally permits the Java syntax for this feature.

The following table shows general differences between Gosu and Java.

General difference	Java	Gosu	Required change?
Omit semicolons in most code. Gosu supports the semicolon, but standard coding style is to omit it. One exception is in block declarations with multiple statements.	`x = 1;`	`x = 1`	Optional.

Print to console by using the `print` function. For compatibility with Java code that is converted to Gosu, you can optionally call the Java class `java.lang.System` print methods.	`System.out.println("hello world");`	Gosu style:
		`print("hello world")`
		Java compatibility style:
		`System.out.println("hello world")`
			Optional.

For Boolean operators, optionally use more natural English versions. Gosu also supports the symbolic versions that Java uses.	`(a && b) || c`	Gosu style:
		`(a and b) or c`
		Java compatibility style:
		`(a && b) || c`
			Optional.

Null-safe property accessor paths.	`// Initialize to null`
	`ArrayList<Windows> x = null`
	`// Check earlier parts of`
	`// path for null, to avoid`
	`// null pointer exception`
	`if (house != null and house.Walls != null) {`
	`  x = house.Walls.Windows`
	`}`
		`var x = house.Walls.Windows`	Optional. Do not make this change for cases in which you rely on throwing null pointer exceptions.

The following table shows the differences in functions and variables between Gosu and Java.

Function or variable difference	Java	Gosu	Required change?

In function declarations:
• Use the keyword `function`.
• List the type after the name, delimited by a colon, for parameters.
• List the type after the parentheses that contain the parameters, delimited by a colon, for the return value.
	`public int addNumbers(int x, String y) {`
	`...`
	`}`
		`public function addNumbers(x : int, y : String) : int {`
		`...`
		`}`
			Required

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

To declare variable argument functions, also called `vararg` functions, Gosu does not support the special Java syntax that specifies the arguments with `...` after the parameter type to indicate a variable number of arguments. Instead, design methods to use arrays or lists.	`public String format(Object... args);`
	`public String errorMessage(Object... args);`
		Method call using array initializer syntax:
		`var c = format({"aa","bb"})`
		Function declaration:
		`public function format(args : Object[]) : String`
		Method call with no arguments:
			Required

Function or variable difference	Java	Gosu	Required change?
To call variable argument Java functions, pass an array of the declared type. To pass no values for the optional parameter, you can pass either an empty array, {} or no value. Gosu array initialization syntax is useful for calling these types of methods.		`var e = errorMessage()`	

Gosu supports the unary operator assignment statements ++ and --, with the following restrictions:
• Only use the operator after the variable, such as i++ .
• These operators form only statements, not expressions.
Gosu supports other compound assignment statements, such as +=, -=.
	`if (++i > 2) {`
	`  ...`
	`}`
		`i++`
		`if (i > 2) {`
		`  ...`
		`}`
			Required

The following table shows the differences in the type system between Gosu and Java.

Type system difference	Java	Gosu	Required change?

For coercions, use the as keyword. Optionally, for compatibility, Gosu supports Java-style coercion syntax.	`int x = (int) 2.1;`	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 by using typeis, which is similar to the Java instanceof operator.	`myobj instanceof String;`	`myobj typeis String`	Required.

Gosu downcasts to a more specific type in if and switch statements. Omit casting to the specific type.	`Object x = "nice";`
	`Int sl = 0;`
	`if (x instanceof String) {`
	`  sl = ((String) x).length;`
	`}`
		`var x : Object = "nice"`
		`var sl = 0`
		`if (x typeis String) {`
		`  sl = x.length // Downcast`
		`}`
			Optional.

To reference the type directly, use typeof. Direct comparisons to a type do not match on subtypes. Consider using 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. This feature is called reified generics. In contrast, Java removes this information. This feature is called type erasure.
From Gosu, Java types lack parameterization even if instantiated in Gosu.
For native Gosu types, Gosu preserves type parameterization at run time.
	`ArrayList<String> mylist = new ArrayList<String>();`
	`system.out.println(typeof mylist);`
	This code prints:
	`ArrayList`
		`var mylist = new ArrayList<String>()`
		`print(typeof mylist)`
		This code prints:
		`ArrayList`
		Note that String is a Java type.
		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)`
			Optional for typical use consuming existing Java types.
			If your code checks type information of native Gosu types, remember that Gosu has reified generics.

Type system difference	Java	Gosu	Required change?
		This code prints: `MyClass<String>`	

Gosu generics support array-style variance of different generic types. In Java, this usage is a compilation error, even though it is natural to assume that it works.	In Java, this code causes 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.

The following table shows the differences in defining classes between Gosu and Java.

Defining classes difference	Java	Gosu	Required change?
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
Declare that you use specific static type members with the keyword `uses` rather than `import`. Use the # symbol before the member name or * symbol, not the period (`.`) symbol.	`import com.abc.MyClass.MY_CONST;` `import com.abc.MyClass.*;`	`uses com.abc.MyClass#MY_CONST` `uses com.abc.MyClass#*`	Required

To declare one or more class constructors, write them as functions named `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

The following table shows the differences in control flow between Gosu and Java.

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, which is an object that implements `Iterable`. Optionally, add `index indexVar` before the closing parenthesis to create a zero-based index variable. If the object to iterate across is `null`, Gosu skips the loop and does not throw an exception. Java does throw an exception in this case.	`int[] numbers = { 1,2,3 };`
	`for (int item : numbers) {`
	`  ...`
	`}`
		`var numbers : int[] = {1,2,3}`
		`for (item in numbers) {`
		`  ...`
		`}`
			Required

The `for` loop syntax in Gosu is different for iterating a loop for an integer number of times. The loop variable contains the zero-based index.
Gosu has native support for intervals, which are sequences of values of the same type between a given pair of end-point values. For example, the set of integers beginning with 0 and ending with 10 has the shorthand syntax `0..10`. Intervals are useful for writing concise `for` loops.
Gosu does not support the `for (initialize;compare;increment)` syntax that Java uses. You can duplicate this syntax by using a `while` statement.
	`for (int i=1; i<20; i++){`
	`  ...`
	`}`
		`for (item in 20) {`
		`  ...`
		`}`
		Using Gosu intervals:
		`for (i in 1..20) {`
		`  ...`
		`}`
			Required

Control flow difference	Java	Gosu	Required change?

		Verbose style:
		`var i = 0`
		`while (i < 20) {`
		`  ...`
		`  i++`
		`}`	

The following table shows optional enhancements that Java does not support and that Gosu provides.

Difference	Java	Gosu	Required change?

Gosu enhancements, which support adding methods and properties to any type, including Java types.	Not applicable	`enhancement StrLenEnhancement : java.lang.String {`
		`  public property get PrettyLength() : String {`
		`  return "length : " + this.length()`
		`  }`
		`}`
			Optional

Gosu blocks, which are in-line functions that act like objects. These blocks are especially useful with the Gosu collections enhancements. Blocks can also be useful as a shortcut for implementing one-method interfaces.	Not applicable	`\ x : Integer -> x * x`	Optional
Native XML support and XSD support.	Not applicable	`var e = schema.parse(xmlText)`	Optional
Native support for consuming web services with syntax similar to native method calls.	Not applicable	`extAPI.myMethod(1, true, "c")`	Optional
Native `String` templates and file-based templates with type-safe parameters.	Not applicable	`var s = "Total = ${ x }."`	Optional

Gosu uses the Java-based collections APIs but improves upon them:
• Simplified initialization syntax that preserves type safety.
• Simpler array-like syntax for getting and setting values in lists, maps, and sets.
• Additional 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. You can call the more verbose Java style for compatibility.
	`List<String> strs = new ArrayList<String>( Arrays.asList( "a", "ab", "abc" ));`	Simplified initialization:
		`var strs = {"a", "ab", "abc"}`
		Array-like "set" and "get":
		`strs[0] = "b"`
		`var val = strs[1]`
		New APIs on Java collections types:
		`strList.each(\ str -> { print(str) })`
			Optional

List and array expansion operator.	Not applicable	`names*.length`	Optional

See also

- “Property accessor paths are null safe” on page 28
- “Variable assignment” on page 103
- “Basic type checking” on page 403
- “Enhancements” on page 201
- “Blocks” on page 175
- “Blocks as shortcuts for anonymous classes” on page 181
- “Working with XML in Gosu” on page 253
- “Templates” on page 301
- “Collections” on page 183
- “List expansion (*.)” on page 187
- *Integration Guide*

Get ready for Gosu

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 readable, powerful, and type-safe type code built on top of the Java platform.

To integrate with external systems, you can use native web service and XML support built directly into the language. You can work with XSD types or external APIs like native objects.

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

The next step for you is to write your first Gosu program and become familiar with the Guidewire Studio application and the Gosu editor in Studio.

See also

- *Configuration Guide*

chapter 2

Gosu types

The Gosu language provides primitive data types and standard object types. The Gosu language also supports the use of Java types.

See also

- "Type system" on page 403

Overview of access to Java types

Gosu provides full access to Java types. Gosu is built on top of the Java language and runs within the Java Virtual Machine (JVM). Gosu loads all Java types, so you have full direct access to Java types, such as classes, libraries, and primitive (non-object) types. You can use your favorite Java classes or libraries directly from Gosu with the same syntax as for native Gosu objects.

The most common Java object types retain their fully qualified name but are always in scope, so you do not need to fully qualify their names in typical code. Examples of classes that are always in scope are `Object`, `String`, and `Boolean`. If your code has ambiguity because of similarly named types also in scope, you must fully qualify these types, such as `java.lang.String`. Other Java types are available to Gosu code, but must be fully qualified, for example `java.io.DataInputStream`.

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

```
var list = new ArrayList<String>()
list.add("Hello Java, from Gosu")
```

Gosu includes transformations on Java types that make your Gosu code clearer, such as turning getters and setters into Gosu properties.

Access to Java types from Gosu includes:

- Instantiation of Java classes with the `new` keyword
- Implementing Java interfaces
- Manipulating Java objects as native Gosu objects
- Calling object methods on instantiated objects
- Exposing object methods that look like getters and setters as properties
- Calling static methods on Java types
- Providing built-in extensions and improvements of many common Java types by using Gosu enhancements
- Support for your own extensions of Java types and interfaces
- Support for Java primitive types

See also

- "Packages always in scope" on page 116
- "Overview of enhancements" on page 23
- "Calling Java from Gosu" on page 139

Primitive Gosu 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 perform better in terms of performance and space compared to their object versions.

The following table compares primitive types and object types.

Type	Primitive types	Object types
Extends from `Object` class		•
Can reference an object		•
A variable of this type could contain `null` at run time		•
Exposes methods		•
Exposes properties		•
Can 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.

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

Primitive	Type	Value
short	Short integer	16-bit two's complement

See also

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

Gosu objects

The root type for all object types in Gosu is the Java class `java.lang.Object`. An object encapsulates some data as variables and provides properties and methods, which are functions that act on the object. Because `java.lang.Object` is always in scope, your code uses `Object` unless a similarly named type is in scope and requires disambiguation.

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.

Do not use the root object type `Object` to create objects directly. In some cases, you need to declare a variable that uses the type `Object` to support a variety of object subclasses. For example, you can define a collection to contain a heterogeneous mix of object types, all of which extend the root object type `Object`.

Example

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

See also

- "Bundles and database transactions" on page 387
- *Configuration Guide*

Object instantiation in Gosu

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 unique instance of the object.

In PolicyCenter, entity instances are special objects defined through the data model configuration files. PolicyCenter persists data in a database at special times in the object life cycle.

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

Example

For example:

```
new java.util.ArrayList() // Create an instance of an ArrayList.
new Integer[5]            // Create an array of integers.
new LossCause[3]          // Create an array of loss cause typecodes.
```

See also

- "Bundles and database transactions" on page 387
- "Using the operator new in object expressions" on page 86
- *Configuration Guide*

Object property assignment

Property assignment is similar to variable assignment in Gosu.

Syntax

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

Some properties are write-protected. Consider the following Gosu code:

```
Activity.UpdateTime = "Feb 17, 2017"
```

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 triggering instantiation of intermediate objects

In general, if you try to assign a property on a variable and the variable evaluates to `null` at run time, Gosu throws a null pointer exception. However, if you set a property by using property path syntax that contains at least two objects, you can instruct Gosu to automatically instantiate an intermediate object in the path.

For example, suppose the following conditions hold:

- 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.
- Your code has a variable called `a` that contains an instance of type `AClass`.

At run time, if `a.B` contains a non-null reference, you can 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"`.

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 the new instance of `BClass`.

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 annotation:

```
@Autocreate
```

See also

- “Handling null values in expressions” on page 96
- “Annotating a class, method, type, class variable, or argument” on page 209

Object property access

Gosu retrieves a property’s value by using the period operator. You can chain this expression with additional property accessors. Gosu handles `null` values in the expression to the left of the period differently from many other languages. Your code uses Gosu behavior to handle `null` values to avoid null pointer exceptions (NPE).

Syntax

```
object.PROPERTY_NAME
```

Examples

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

See also

- “Handling null values in expressions” on page 96

Object methods in Gosu

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 parentheses after the name to denote the function. Gosu uses the dot notation to call a method on a object instance.

Syntax

```
object.METHOD_NAME()
```

Example

Expression	Result
`claim.isClosed()`	`Returns a Boolean value indicating the status of Claim`
`claim.resetFlags()`	`Resets flags for this claim`

See also

- “Handling null values in expressions” on page 96
- “Static method calls” on page 95

Using Boolean values in Gosu

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 `java.lang.Boolean` is always in scope, your code uses `Boolean` unless a similarly named type is in scope and requires disambiguation.

For both `boolean` and `Boolean`, Gosu coerces values to `true` or `false`. The following table describes coercion rules.

Value	Type of value	Coerces to	Note
`1`	`int`	`true`	
`0`	`int`	`false`	
`"true"`	`String`	`true`	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"`.
`"false"` or any non-null `String` value other than `"true"`	`String`	`false`	See note for column value `"true"`.
`null`	See note	If coerced to the `boolean` type, the result depends on the 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`.	The `boolean` type is a primitive and can never contain `null`. The `Boolean` type is an object type, and variables of that type can contain `null`. Be careful to check for `null` values for variables declared as `String`. A `null` value might indicate uninitialized data or other unexpected code paths.

Notice the following 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.

Be careful to check for `null` values for variables declared as `String` to avoid ambiguity in your code. A `null` value might indicate uninitialized data or other unexpected code paths.

Example

```
var hasMoreMembers : Boolean = null
var isDone = false
```

Using characters and String data in Gosu

To represent a single character in Gosu, use the primitive type `char`. To type a single character literal, surround it with single quotes, such as `'x'`.

To represent a sequence of characters in Gosu, use the Java type `java.lang.String`. Because `java.lang.String` is always in scope, your code uses `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). If you need to type a quotation mark character in a `String` literal, escape the quotation mark character by putting a backslash before it, such as `"hello \"Bob\""`.

Alternatively, you can use single quotation marks instead of double quotation marks. This style is useful if you want to type `String` literals that contain the quotation mark character because you do not need to escape quotation mark characters:

```
var c = 'Hello "Bob"'
```

If you use single quotes and the literal has exactly one character, Gosu infers the type to be `char` instead of `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("s1 has content = " + s1.HasContent)
print("s2 has content = " + s2.HasContent)
print("s3 has content = " + s3.HasContent)

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

This code prints:

```
s1 has content = false
s2 has content = false
s3 has content = true
s1 is empty = false
s2 is empty = true
s3 is empty = false
```

If the variable holds an empty `String` or `null`, the `HasContent` method returns `false`. Using the `HasContent` method is more intuitive than using the `Empty` property in typical cases where `null` represents absence of data.

Setting values of String properties in entity instances

Setting a value on a `String` data type property on an entity instance causes 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. This behavior is configurable.
- If the result is the empty `String` (`""`), Gosu sets the value to `null` instead of the empty `String`. This behavior 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 code prints:

```
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.

Configuring whitespace removal for entity text properties

You can control whether PolicyCenter 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 PolicyCenter from trimming whitespace before committing a `String` property to the database, include the `trimwhitespace` column parameter in the column definition, and set the parameter to `false`. The XML definition for a column that does not trim whitespace looks like the following lines:

```
<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 PolicyCenter trims leading and trailing spaces. You cannot configure whether PolicyCenter coerces an empty string to `null`.

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

See also

- “String variables can have content, be empty, or be null” on page 59
- “Setting values of String properties in entity instances” on page 59

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 code 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 the API class `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, to perform search and replace, instead of the Java native method `replace` on `java.lang.String`, 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 functionality is called a template. For example, the following String assignment uses template features:

```
var str = "One plus one equals ${ 1 + 1 }."
print(str)
```

This code prints:

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

See also

- “Templates” on page 301

Escaping special characters in strings

In Gosu strings, the backslash character (\) indicates that the immediately following character requires special handling. Because the backslash 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 quotation mark into a string without terminating the string. 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 escape sequence is not used in embedded code inside Gosu templates. In such cases, do not escape the quotation mark characters.
`\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\""
```

See also

- “Templates” on page 301

Gosu String templates

In addition to simple text values enclosed by quotation marks, you can embed Gosu code in the definition of a `String` literal. At compile time, Gosu uses its native type checking to ensure that the embedded Gosu code is valid and type safe.

Embedding a Gosu expression into text

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

```
${ EXPRESSION }
```

If the expression does not return a value of type `String`, Gosu attempts to coerce the result to the type `String`.
For example, suppose you need to display text with a calculation in the middle of the text:

```
var mycalc = 1 + 1
var str = "One plus one equals " + mycalc + "."
print(str)
```

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

```
var str = "One plus one equals ${ 1 + 1 }."
print(str)
```

This code prints:

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

Embedding Gosu statements into text

To embed one or more statements, which do not return a value, in `String` text, you use a template scriptlet. Use the two-character text `<%` to begin the scriptlet. Use the two-character text `%>` to end the scriptlet:

```
<% STATEMENT LIST %>
```

Gosu runs the statements in the scriptlet before evaluating the text in the string.
The following code shows the use of scriptlets in a `String` value:

```
var str = "<% for (i in 1..5) { var odd = (i % 2 == 1) %>${i} is ${ (odd?"odd":"even") }\n<% } %>"
print(str)
```

This code prints:

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
```

Quotation marks in templates

Within a code expression or statement inside a template, do not escape quotation mark characters by using a backslash character. The following line is valid Gosu code:

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

The following line is invalid because of incorrect escaping of the internal quotation marks:

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

See also

- "Templates" on page 301

Entity types

An entity is 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.

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. 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 get the type from an entity dynamically, for example to get the subtype of an entity instance. If you have a reference to an entity instance, get its `Type` property. For example, if you have a variable `ad` that contains an `Address` entity instance, use code such as the following:

```
myClass.myMethod(ad.Type)
```

WARNING The base configuration includes visible usages of the entity instance method `setFieldValue`. The `setFieldValue` method is reserved for Guidewire internal use. Calling this method can cause serious data corruption or errors in application logic that may not be immediately apparent. You must obtain explicit prior approval from Guidewire Support to call `setFieldValue` as part of a specific approved workaround.

See also

- *Configuration Guide*

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 `Policy` entity instance (`Policy.Id`) has type `Key`. Casting an entity instance to a key supports functions or internal domain methods that use `Key` objects as parameters.

Example

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

Do not confuse a key (the type called `Key`) and a typekey (the type called `Typekey`).

See also

- "Typekeys and typelists" on page 64

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 also known as a typecode.

For example, the typelist called `AddressType` contains typekey values `BUSINESS`, `HOME`, and `OTHER`, which are available as constants on the typelist type with some changes to the name.

Typelist Literals

In most cases, to get a literal for a typelist, you can type the typelist name in the appropriate programming context. Ambiguity about the name or package of the typelist might occur in some programming contexts. To resolve this ambiguity, type the fully qualified syntax `typekey.`*`TYPELISTNAME`*. For example, `typekey.AddressType`.

Typecode Literals

To reference an existing typecode from Gosu, access the typecode by using the typelist name and the typecode value in capital letters and the prefix `TC_`:

```
TYPELISTNAME.TC_TYPECODENAME
```

For example, to select from the `AddressType` typelist a reference to the typekey with code `BUSINESS`, use:

```
AddressType.TC_BUSINESS
```

In the data model configuration, the typecode definition has an optional `identifierCode` attribute, which changes how the application generates the programmatic typekey name in Gosu. The value of the `identifierCode` attribute can include only characters that are valid for a Gosu identifier such as a class or variable name. For example, you cannot include a hyphen (`-`) character in the value of the `identifierCode` attribute. The maximum length of the identifier code value is three characters fewer than the maximum length of a Gosu identifier name. For practical purposes, this length is unlimited.

PolicyCenter uses the following rules to generate the typecode literal:

- If the optional `identifierCode` attribute exists, PolicyCenter transforms the `identifierCode` attribute to upper case, and adds the prefix `TC_`. Using the `identifierCode` attribute ensures that all identifier names are unique, which prevents name conflicts among typecode literals.
- If the `identifierCode` attribute does not exist for a typecode, PolicyCenter generates the typecode literal from the `code` attribute. PolicyCenter transforms the `code` attribute to upper case, replaces invalid characters with underscores (_), and adds the prefix `TC_`. For example, codes `abc`, `123`, and `a-b` become the identifiers `TC_ABC`, `TC_123`, and `TC_A_B`, respectively. The substitution of invalid characters by underscores might cause two codes to become the same identifier. For example, codes `A_B` and `A-B` both become `TC_A_B`, which is a conflict that causes compilation errors.

Getting 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 both retired and 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 obsolete typekeys that are marked as retired. If the argument is set to `true`, the return value includes retired typekeys. Otherwise, the method returns 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.AllTypeKeys[0].Code)
```

This code prints:

```
[Billing, Business, Home, Other]
billing
```

Getting information from a typekey

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

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

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
```

Gosu array types

An *array* in Gosu is a collection of data values or objects of the same type. A number, or *index*, identifies the position of each element in the array. Index values for arrays begin with zero and increment by one. The index value `0` identifies the first element of an array.

Creating a Gosu array

Use square brackets after a type name to create an array. Gosu constrains elements of the array to values or objects of that type. For example, the expression `new String[]` creates an array of `String` elements.

Whenever you create an array, you must set its size in terms of a fixed number of elements. You can set the size explicitly by including the number of elements within the square brackets. The following statement explicitly sets the size of a `String` array to four elements.

```
var stringArray = new String[4]
```

Alternatively, you can set the size of an array implicitly by providing a list of values or objects with which to initialize the array. The size of the array is the number of items in the list. The following statement implicitly sets the size of a `String` array to four elements.

```
var stringArray = new String[] {"one", "two", "three", "four"}
```

Initializing a Gosu array with default values

You can create an array with all of its elements set to the same default value by calling the `makeArray` Gosu enhancement method on the `java.utils.Arrays` class. Pass the default value and the size of the array. Gosu sets the type of the array to the same type as the default value. The following statement creates a `String` array of five elements, with each element initialized to the value `no`.

```
var stringArray = java.util.Arrays.makeArray("no", 5)
```

Accessing elements of a Gosu array

In Gosu, you access elements of an array with array index notation. Place the index value for the element you want to access within square brackets that follow the array name. Index values for array elements begin with zero and increment by one. The following code uses the index value `2` to access the third element of an array.

```
// Create an array with four elements.
var stringArray = new String[] {"one", "two", "three", "four"}

// Print the the third element of the array.
print ("Third array element contains the value " + stringArray[2] + ".")
```

The output from this code is:

```
The third array element contains the value three.
```

Examples

The following code accesses the first exposure on a claim.

```
Claim.Exposures[0]
```

The following code creates an array of five elements and then accesses the third element, the character “c”.

```
gw.api.util.StringUtil.splitWhitespace( "a b c d e" )[2]
```

Iterating the elements of a Gosu array

In Gosu, you can iterate through the elements of an array in a `for` loop. You declare a temporary iteration variable to hold an element of the array during each loop iteration, and you specify the name of the array through which to iterate. The following code declares the iteration variable `element` and then uses it to print each element of the array.

```
var stringArray = new String[] {"one", "two", "three", "four"}

for (element in stringArray) {
  print(element)
}
```

The output from this code is:

```
one
two
three
four
```

See also

- "Iteration in for statements" on page 107

Gosu arrays and the expansion operator

Gosu includes a special operator for array expansion and list expansion. The expansion operator is an asterisk followed by a period. Array expansion is valuable if you need a single one-dimensional array or list through which you can iterate.

The return value is as follows:

- If you use the expansion operator on an array, Gosu gets a property from every item in the array and returns all instances of that property in a new, single-dimensional, read-only array.
- If you use the expansion operator on a list, Gosu gets a property from every item in the list and returns all instances of that property in a new, single-dimensional, read-only list.

You can access elements in the new array or list individually with index notation, and you can iterate through the elements in a `for` loop. You cannot modify the elements of the new array or list.

The following code creates a `String` array. String objects in Gosu have a `length` property that contains the number of characters. The code uses the expansion operator on the array to create an array of string lengths for elements of the original array. Because the `length` property is of type `int`, the expansion array also is of type `int`.

```
var stringArray = new String[] {"one", "two", "three", "four"}
var lengthArray = stringArray*.length

for (element in lengthArray) {
  print(element)
}
```

The output from this code is:

```
3
3
5
4
```

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.

Suppose you wanted to get a list of the groups to which a user belongs so that you can display the display name property of each group. Suppose a `User` object contains a `MemberGroups` property that returns a read-only array of

groups to which the user belongs. In this case, the Gosu syntax `user.MemberGroups` returns an array of `Group` objects, each one of which has a `DisplayName` property. To get the display name property 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 also applies to methods. Gosu uses the type on which the method runs to determine how to expand the type:

- If the original object is an array, Gosu creates an expanded array.
- If the original object is a list, Gosu creates an expanded list.

The following example calls a method on the `String` component of the `List` of `String` objects. Gosu generates a list of initials by extracting the first character in each `String`.

```
var s = {"Fred", "Garvin"}

// Generate the character list [F, G]
var charList = s*.charAt( 0 )
```

Important notes about the expansion operator:

- The generated array or list itself is always read-only from Gosu. You cannot assign values to elements of the array, such as setting `nameArray[0]`.
- The expansion operator `*.` applies only to array expansion, not for standard property accessing.
- When using the `*.` expansion operator, only component type properties are accessible.
- When using the `*.` expansion operator, array properties are not accessible.
- The expansion operator applies to arrays and to any `Iterable` type and all `Iterator` types. The operator preserves the type of the array or list. For instance, if you apply the `*.` operator to a `List`, the result is a `List`. All other expansion behavior is the same as for arrays.

See also

- “Enhancements on Gosu collections and related types” on page 189

Array list access with array index notation

In Gosu, you can access elements of Java language array lists by using array index notation. You can iterate through Java array lists in a `for` loop the same as a Gosu array. The following code creates a Java array list, populates the list, and updates one element in the list by using array index notation. Then the code uses a `for` loop to iterate through the elements of the list and print them.

```
// Create a Java array list.
var list = new java.util.ArrayList()

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

//Assign a value to a element of the Java list.
list[3] = "threeUPDATED"

//Iterate through the Java list the same way as a Gosu array.
for (element in list) {
```

```
  print(element)
}
```

The output for this code is:

```
zero
one
two
threeUPDATED
four
```

In many situations, using collection classes, such as `java.util.ArrayList` or `java.util.HashMap`, is a better choice than using Gosu arrays. For example, the sizes of lists and maps grow and shrink as you add and remove members from them. The `ArrayList` and `HashMap` classes are the native Java array lists and hash maps. Gosu adds enhancement methods to these types or their parent classes/interfaces, such as the interface `java.util.List`.

See also

- “for statements” on page 107
- “Collections” on page 183

Associative array syntax for property access

For most types other than Gosu arrays, you can use associative array syntax to access the elements, or properties, of an object. With *associative array syntax*, you access an object property with a `String` value in square brackets that contains the property name instead of an index number.

The following code uses associative array syntax to get the postal code of a contact.

```
contact["PostalCode"]
```

Use associative array syntax if a property name is unknown at compile but can be determined at run time with input from users. If the property name does not exist at run time, Gosu throws an exception.

> **WARNING** Gosu cannot check associative array syntax at compile time to be certain that a property name is accurate or that the property exists. Be careful whenever you use associate array syntax to catch unexpected run time errors. Use property path expressions, which provide type-safe property access, instead of associative array syntax whenever possible.

Associative arrays on objects in Gosu are similar to instances of the Java map class `java.util.Map`. Associative array syntax for property access works with most classes, including the `Map` class and types that do not take array-style index numbers.

You cannot use associative array syntax with `String` objects in Gosu. A `String` object behaves like an ordered list of characters and requires array index notation to access its individual character elements.

Examples

The following code demonstrates single associative array syntax to get the `StreetAddress` property on a `Person` object. At run time, the code is equivalent to the property path expression `person.StreetAddress`. The code might evaluate to `123 Main Street`.

```
person["StreetAddress"]
```

The following code demonstrates double associative array syntax to get the `City` property on the `Address` object that is a property of a `Person` object. At run time, the code is equivalent to the property path expression `person.Address.City`. The code might evaluate to `Birmingham`.

```
person["Address"]["City"]
```

The following code uses single associative array syntax to set the `City` property on an `Address` object.

```
newAddress["City"] = "Birmingham"
```

See also

- "Null-safe property access" on page 96
- "Collections" on page 183

Entity arrays

In the data model definition of an entity type, you can define a property that is an array of read-only entity instances of another specific type.

By default, array fields on entities are unordered indexed arrays in Gosu. An indexed entity array acts like a typical array in that you can get an object by using a numeric index offset from the beginning. For example, `myArray[2]`. Because the items are unordered, do not rely on the meaning or long term position of a specific element. However, use numeric ordering to loop across the array with looping syntax, such as a `for` loop.

Gosu defines other enhancement methods and properties on arrays, such as `each`, `firstWhere`, `HasElements`, and `map`. Use enhancement methods and properties to write concise powerful Gosu code. However, be aware that bringing many entities into memory can be resource intensive if the entity array size is large.

For example, in PolicyCenter, on a `User` entity type definition file in `user.eti`, the property `Roles` is an array of `role` objects that relate to a user:

```
<array arrayentity="UserRole" name="Roles" ... />
```

Gosu adds an `addTo…` method and a `removeFrom…` method to the containing entity type to help manage the elements of each associative array defined on the entity. In the preceding example, Gosu adds the methods `addToRoles` and `removeFromRoles` to the entity type `User`.

Optionally, entity arrays support an associative syntax like a map.

Entity array addTo… methods

The `addTo…` method sets a foreign key on each element in an entity array that points back to the entity that owns the array. You must call the `addTo…` method on newly created array elements. Otherwise, the Rule Engine does not commit the array elements to the database. As a best practice, Guidewire strongly recommends that you always call the appropriate `addTo…` method to relate new array elements with the entity that owns the array.

IMPORTANT Entity arrays are inherently unordered with a non-deterministic order each time they are generated. For example, the order in which an `addTo…` method adds elements to its entity array is not necessarily the order in which a `for` loop retrieves them.

Entity array removeFrom… methods

The `removeFrom…` method behaves differently depending on whether the elements of an entity array are a retireable type:

- If the element type of the array is retireable, the `removeFrom…` method retires the element.
- If the element type of the array is not retireable, the `removeFrom…` method deletes the element.

The following example removes elements in an array in which the primary contributing factor was driver error.

```
for (factor in claim.ContributingFactors) {
  if (factor.Primary == "Driver Factors") {
    claim.removeFromContributingFactors(factor)
  }
}
```

Optional associative entity arrays

By default, entity arrays are unordered indexed arrays. Optionally, you can define an entity array to support associative array syntax. An associative array acts like a container with key-value mapping similar to `java.util.HashMap`. Get items from an associative array by providing a key instead of a numeric index. For example, `myArray["en_US"]`. In the data model definition files, you can make associative arrays by adding the `<array-association>` subelement to the `<array>` element.

The modified array field still behaves as an indexed array but you can optionally get items by using associative array syntax.

See also

- "Enhancements on Gosu collections and related types" on page 189
- *Configuration Guide*

Casting arrays

The `as` keyword can be used to cast an array type to another compatible array type. For example, `String` is a subtype of `Object`, so an array of type `String` can be cast to an array of type `Object`.

```
var objArray : Object[]
objArray = new String[] {"a", "b"} as Object[]   // Okay to cast String[] to Object[]
```

The reverse operation of casting an `Object` to a `String` is invalid, as demonstrated below.

```
var objArray = new Object[1]
objArray[0] = "foo"
var strArray = objArray as String[]   // Run-time error attempting to cast Object[] to String[]
```

The compiler can detect and produce a compilation error when two referenced types are incompatible. For example, an `Integer[]` can never be converted to a `String[]`, so the following code statements produce a compilation error.

```
var strArray : String[]
strArray = new Integer[] {2, 3} as String[]   // Compilation error attempting to cast Integer[] to String[]
```

In some cases, you know that all the objects in an array are of a particular subtype of the defined type of the array. In other cases, you filter an array to retrieve only objects of a particular subtype. As shown in a preceding code example, you cannot use the `as` keyword to convert an array to an array of a subtype. Instead, you can use the Gosu enhancement method `cast` to make an array of the subtype. You can then perform operations on the new array that are specific to the subtype and not available to the parent type. The following code demonstrates how to use the `cast` method.

```
var objArray = new Object[1]
objArray[0] = "foo"
var strArray = objArray.cast(String)

var objArray2 = new Object[2]
objArray2[0] = "foo"
objArray2[1] = 42
var strArray2 = objArray2.where(\aid -> aid typeis String).cast(String)
```

You cannot use the `cast` method to convert incompatible types. The compiler does not detect this type of error. The following code produces a run-time error.

```
var strArray : String[]
strArray = new Integer[] {2, 3}.cast(String)   // Run-time error attempting to cast Integer[] to String[]
```

Numeric literals in Gosu

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

For example:

- Gosu infers that the following variable has type `BigInteger` because the right side of the assignment uses a numeric literal `4bi`. That literal means "4, as a big integer".

```
var aBigInt = 4bi
```

- 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 = 4f
var anotherFloat = 4.0f
```

You can omit the suffix of a numeric literal if the type is declared explicitly such that no type inference is necessary.

Gosu does not support floating point hexadecimal literals.

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

Type	Suffix or prefix	Examples
byte	suffix: b or B	`var aByte = 4b`
short	suffix: as short	`var aShort = 4 as short`
int	none	`var anInt = 4`
long	suffix: l (lowercase L) or L	`var aLong = 4L`

float	suffix: f or F	`var f1 = 4f`
		`var f2 = 4.0f`
		`var f3 = 4.0e3`

double	suffix: d or D	`var aDouble = 4d`
BigInteger	suffix: bi or BI	`var aBigInt = 4bi`
BigDecimal	suffix: bd or BD	`var aBigD = 4bd` `var anotherBigD = 4.0bd`

int as binary or mask	prefix: 0b or 0B	`var maskVal1 = 0b0001  // 0 and 1 only`
		`var maskVal2 = 0b0010`
		`var maskVal3 = 0b0100`

int as hexadecimal	prefix: 0x or 0X	`var aColor = 0xFFFF // 0 through F only`

Scientific notation and floating point

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

For example, consider the number: 1.23 * 10 to the 11th power.

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

You can use the scientific notation anywhere you can use a `float` literal or `double` literal.

Examples

```
var result1 = 9.2 * 3
var result2 = 2.057e3
```

```
print (result1)
print (typeof result1)

print (result2)
print (typeof result2)
```

This code prints:

```
27.599999999999998
double
2057.0
double
```

Compatibility with earlier Gosu releases

Gosu provides an unsupported type that exists for compatibility with earlier Gosu releases.

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[]`.

See also

- “Gosu array types” on page 65

chapter 3

Operators and expressions

Gosu provides many operators that are similar to operators in the Java language. Gosu also provides additional operators. You use the operators, constants, and variables to create expressions.

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 ternary operator). The following list provides examples of each operator type:

Operator type	Arguments	Examples of this operator type
unary	1	• `-3` • `typeof "Test"` • `new ArrayList<Address>()`
binary	2	• `5 - 3` • `a and b` • `2 * 6`
ternary	3	• `3*3 == 9 ? true : false`

See also

- “Operator precedence” on page 75

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.

`new`	Object creation, object reflection.
`,`	Array value list, as in {*value1*, *value2*, *value3*} Argument list, as in (*parameter1*, *parameter2*, *parameter3*)
`as`	Casting a value to a specific type.
`+` `-`	Unary operands for positive value and negative value.

`~`
`!`
`not`
`typeof`
`eval`
	Bit-wise `OR`, logical `NOT`, type of, `eval(`*`expression`*`)`.

`typeis`	Determine if an object is a specific type or subtype.

`*`
`/`
`%`
	Multiplication, division, modulo division.

`<<`
`>>`
`>>>`
	Bitwise shifting.

`+`
`-`
`?+`
`?-`
	Addition, subtraction, string concatenation. The versions with the question marks are the null-safe versions.

`<`
`<=`
`>`
`>=`
	Less than, less than or equal, greater than, greater than or equal

`==`
`===`
`!=`
`!==`
	Equality and inequality operators for values and references.

`&`	Bitwise `AND`.
`^`	Bitwise exclusive `OR`.
`|`	Bitwise inclusive `OR`.
`&&` `and`	Logical `AND`. The two variants are equivalent.
`||` `or`	Logical `OR`. The two variants are equivalent.
`? :`	Ternary conditional. For example: `3*3 == 9 ? true : false`

Operator	Description
`= += -= *= /= %= &= &&= ^= |= ||= <<= >>= >>>=`	Assignment operator statements. These operators apply to Gosu statements, not expressions.

See also

- “Handling null values in expressions” on page 96
- “Null-safe math operators” on page 99
- “Equality expressions” on page 81
- “Gosu variables” on page 102

Standard Gosu expressions

A Gosu expression results in a single value. A Gosu expression is categorized by the type of operator used in constructing it. Arithmetic expressions use arithmetic operators (`+`, `-`, `*`, `/`). Logical expressions use logical operators (`AND`, `OR`, `NOT`). Other topics contain descriptions and examples of Gosu-supported expressions and how to use them. Gosu also supports type-cast expressions and type-checking expressions.

Arithmetic expressions in Gosu

By default, Gosu arithmetic expressions do not check for overflow errors. Optionally you can enable checked arithmetic, which generates Gosu exceptions for overflow errors.

See also

- “Checked arithmetic for add, subtract, and multiply” on page 80

Arithmetic expression types

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. If the left-hand or right-hand operand of the operator is a Java `BigDecimal` or `BigInteger`, the result is `BigDecimal` or `BigInteger` also. This coercion is especially important for accuracy, which currency values often require.

Addition and concatenation operator (+)

The `+` operator performs arithmetic addition or string concatenation using either two numeric data types or two `String` data types as operands. If you add two numeric types, the result is numeric. If you add two `String` objects, the result is a `String`. Note the following:

- If both operands are numeric, the `+` operator performs addition on numeric types.
- If either operand is a `String`, Gosu converts the operand that is not a `String` to a `String`. The result is the concatenation of the two strings.

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

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

IMPORTANT By default, the addition operator does not check for overflow errors. Optionally, you can enable checked arithmetic, which generates Gosu exceptions for overflow errors.

Subtraction operator (-)

The `-` operator performs arithmetic subtraction, using two numeric values as operands. The result is the same type as the input types.

Expression	Result
`9 - 2`	`7`
`8 - 3.359`	`4.641`

IMPORTANT By default, the subtraction operator does not check for overflow errors. Optionally, you can enable checked arithmetic, which generates Gosu exceptions for overflow errors.

Multiplication operator (*)

The * operator performs arithmetic multiplication, using two numeric values as operands. The result is the same as the input types.

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

IMPORTANT By default, the multiplication operator does not check for overflow errors. Optionally, you can enable checked arithmetic, which generates Gosu exceptions for overflow errors.

Division operator (/)

The `/` operator performs arithmetic division using two numeric values as operands. The result is the same as the input types. The result of floating-point division follows the specification of IEEE arithmetic.

Expression	Result
`10 / 2`	`5`
`1 / 0`	`Infinity`
`0 / 0`	`NaN`
`0/1`	`0`

Arithmetic modulo operator (%)

The `%` operator performs arithmetic modulo operations, using numeric values as operands. The result is the same type as the input types. The result of a modulo operation is the remainder if the numerator divides by the denominator.

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

Bitwise AND (&)

The `&` operator performs a binary bitwise `AND` operation on 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 expression 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 expression 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 3 is 0011 binary. In binary, this does a bitwise OR between value 1010 and 0011. The result is binary 1011, which is decimal 11.

Bitwise exclusive OR (^)

The ^ (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.

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 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 behavior 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 expression does a binary bitwise right shift two bits to the right, filling in the top two bits with a 1 because the original number was negative. The result is binary 11111111 11111111 11111111 11111101, which is decimal -3.

Bitwise 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 for each shift to zero. For unsigned integer values, this behavior is the usually the desirable one. Contrast this behavior 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 highest value bit is set to 0. The result is binary 0101, which is decimal 5.

The expression `-10 >>> 2` evaluates to 1,073,741,821. The decimal number -10 is 11111111 11111111 11111111 11110110 binary. This expression does a binary bitwise right shift two bits to the right, filling in the top two bits with a 0. The result is binary 00111111 11111111 11111111 11111101, which is decimal 1,073,741,821.

See also

- “Null-safe math operators” on page 99
- “Checked arithmetic for add, subtract, and multiply” on page 80

Checked arithmetic for add, subtract, and multiply

By default, numeric values could exceed their defined bounds in arithmetic operations. For example, if you multiplied the maximum integer value by 2, the result by definition exceeds the range of integer values. Not only is the result value incorrect, the result could be positive when you expect it to be negative, or negative when you expect it to be positive. Because no Gosu exceptions occur, writing code to protect against overflow errors, which could cause unexpected behavior or security issues, is difficult.

Gosu includes an optional feature called checked arithmetic. If *checked arithmetic* is enabled, Gosu behavior for the standard arithmetic operators changes for addition, subtraction, and multiplication. In nearly all cases, the result is the same and the behavior is the same. In the rare case that arithmetic overflow occurs, Gosu throws the exception `ArithmeticException`.

To enable checked arithmetic, set Java system property `checkedArithmetic` to `true` when launching Studio and when launching the application server.

Gosu checked arithmetic includes protection only for the operators `+`, `-`, and `*`. There is no protection for division, which only affects the expression `Integer.MIN_VALUE / -1`.

In some cases, arithmetic overflow behaviors are desirable for operators `+`, `-`, and `*`. For example, some common hash algorithms rely on arithmetic overflow. To handle typical use cases in overridden `hashCode` methods, Gosu always compiles `hashCode` methods with checked arithmetic disabled.

For other cases in which arithmetic overflow behaviors are desirable, you can use three Gosu operators that ensure unchecked arithmetic independent of the Java system property `checkedArithmetic`. The new operators are the standard arithmetic operators prefaced with an exclamation point character: `!+`, `!-`, and `!*`.

For example, with checked arithmetic enabled:

```
var four = new Integer(4)
var y = Integer.MAX_VALUE * 2 + four // This line throws ArithmeticException
```

In contrast, the following example uses the unchecked arithmetic operator `!*`, which is only for special circumstances in which overflow is desirable:

```
var four = new Integer(4)
var x = Integer.MAX_VALUE !* 2 + four // This line does not throw ArithmeticException
print(x) // Print "2"
```

Because the arithmetic in the second example is unchecked, the result that prints `2` successfully is possibly unexpected and invalid.

See also

- *Configuration Guide*

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:

- Relational equality operator (`==`)
- Object equality operator (`===`)
- Inequality operator (`!=`)
- Object inequality operator (`!==`)

Relational equality operator (==)

The `==` operator tests for relational equality. The operands can be of any compatible types. The result is always `Boolean`. For object types, the following rules apply in the following order:

- If both objects implement the `Comparable` interface, the `==` operator calls the `Comparable` interface method `compareTo`. Gosu calls the `compareTo` method on the object to the left of the `==` operator.
- Otherwise, the `==` operator calls the native Java method *`object`*`.equals()` to compare values. Gosu calls the `equals` method on the object to the left of the `==` operator.

IMPORTANT To determine whether two objects are the same in-memory object, instead use the `===` or `!==` operators.

In the Java language, the `==` operator evaluates to `true` if and only if both operands have exactly the same reference value, which refers to the same object in memory. This behavior works well for primitive types like integers. For reference types, you usually do not want to compare the object in memory. Instead, to compare value equality, Java code typically uses *`object`*`.equals()`, not the `==` operator. The *`object`*`.equals()` method includes the comparison of object reference values, but also compares property values.

If you use the `==` operator for comparison of reference types, Gosu calls *`object`*`.equals()`. In most cases, this behavior is what you want for reference types.

Syntax

```
a == b
```

Examples

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

Object equality operator (===)

You compare identity references, rather than the object values, to determine whether two objects reference the same in-memory object. You use the Gosu operator `===` (three equal signs) to compare object equality.

The operands can be of any compatible types. The result type is always `Boolean`.

Examples comparing == and ===

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

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

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

`print("3" === "3")`	`true`	This example is harder to understand. By just looking at the code, it seems like these two `String` objects would be different objects. However, the Gosu compiler detects that the objects are the same `String`. Gosu optimizes the code to point to the same `String` object in memory for both usages of the `String` literal `"3"`.
`print("3" === new String("3"))`	`false`	The two `String` objects both contain the value `"3"`, but are not the same object.

Inequality operator (!=)

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

For object types, the following rules apply in the following order:

- If both objects implement the `Comparable` interface, the `!=` operator calls the `Comparable` interface method `compareTo` to compare values. Gosu calls the `compareTo` method on the object to the left of the `==` operator.
- Otherwise, the `!=` operator calls the native Java method *`object`*`.equals()` to compare values. Gosu calls the `equals` method on the object to the left of the `==` operator.

IMPORTANT To compare whether two objects are the same in-memory object, instead use the `===` or `!==` operators.

Syntax

```
a != b
```

Examples

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

Object inequality operator (!==)

The `!==` operator tests for object inequality. It returns the opposite value of the `===` operator, which tests for object equality not value equality.

The operands can be of any compatible types. The result type is always `Boolean`.

Syntax

```
a !== b
```

Examples comparing != and !==

Expression	Output	Description
`print("3" != "4")`	`true`	The two `String` objects contain different values.
`print("3" !== "4")`	`true`	Gosu represents the two `String` literals as separate objects in memory (as well as separate values).

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

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

`print("3" != "3")`	`false`	The two `String` objects contain the same value. Compare to the examples in the following rows.
`print("3" != new String("3"))`	`false`	The two `String` objects contain the same value.
`print("3" !== "3")`	`false`	This example is harder to understand. By just looking at the code, it seems like these two `String` objects would be different objects. However, in this case, the Gosu compiler detects they are the same `String` at compile time. Gosu optimizes the code for both usages of a `String` literal to point to the same object in memory for both usages of the `"3"`.
`print("3" !== new String("3"))`	`true`	The two `String` objects both contain the value `"3"`, but are not the same object.

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

Logical expressions

Gosu logical expressions use standard logical operators to evaluate the expression in terms of the `boolean` values of `true` and `false`.

Gosu supports the following logical expressions:

- Logical `AND`
- Logical `OR`
- Logical `NOT`

Gosu evaluates logical expressions from left to right and uses the following rules to test the expressions for possible short-circuit evaluation:

- `true OR` *`any-expression`* always evaluates to `true` – Gosu only runs and evaluates *`any-expression`* if the expression before the `OR` is false. If Gosu determines that the expression before the `OR` evaluates to `true`, the second 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. If Gosu determines that the expression before the `AND` evaluates to `false`, the second expression is not evaluated.

Logical AND

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

Syntax

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

Examples

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

Logical OR

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

Syntax

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

Examples

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

Logical NOT

To indicate a logical negation (a logical `NOT` expression), use either the keyword `not` or the exclamation point character (`!`). 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 the `NOT` operator.

- **Bad example** – The following example demonstrates how not to use the logical `NOT` operator.

  ```
  if (not liabilityCov.Limit == line.BOPLiabilityCov.Limit) {
    return true
  }
  ```

  This example causes an error at run time because Gosu associates the `NOT` operator with the variable to its right before evaluating the expression. This association causes the expression to become:

  ```
  (false == line.BOPLiabilityCov.Limit)
  ```

  This comparison causes a class-cast exception, as follows:

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

- **Better example** – The following example demonstrates how to use the `NOT` operator correctly.

  ```
  if (not (liabilityCov.Limit == line.BOPLiabilityCov.Limit)) {
    return true
  }
  ```

  In this example, the extra parentheses force the desired comparison, and associate the `NOT` operator with the correct expression.

- **Preferred example** – Use the `!=` operator rather than `NOT` for writing code of this type.

  ```
  if (liabilityCov.Limit != line.BOPLiabilityCov.Limit) {
    return true
  }
  ```

  The expression had no need to use the `NOT` operator. The final code expression is somewhat simpler and does exactly what is required.

typeis expressions

Gosu uses the operator `typeis` to test the type of an object against a named type. The result is `true` if the object has that type or a subtype.

See also

- "Using Boolean values in Gosu" on page 58
- "Test type with typeis" on page 404

Using the operator new in object expressions

Gosu uses the `new` operator to create an instance of a type. The type can be a Gosu class, a Java class, a Guidewire entity type, or an array. Creating a new instance is also called *instantiation*. Instantiation allocates memory to a new object of that type and initializes the object. After you instantiate an object, you can call object methods or access instance fields and properties.

At least one constructor (creation function) must be exposed on a type to construct an instance of the type with the `new` operator. You can have multiple constructors with different types and numbers of arguments.

Although creating a new object is typically used as an expression, it can also optionally be a statement if the return value is not needed.

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 passed as arguments to determine which constructor to call.

Examples

Expression	Result
`new java.util.HashMap(8)`	Creates an instance of the `HashMap` Java class.
`new String[12]`	Creates a `String` array that has12 members with no initial values.
`new String[] {"a", "b", "c"}`	Creates a `String` array that has three members, initialized to `"a"`, `"b"`, and `"c"`.

See also

- "Using the operator new as a statement" on page 102

Special behaviors for entity types during instantiation

There are special behaviors for instantiating Guidewire entity types, which are Gosu types that you define by editing data model XML configuration files.

Required Fields During Instantiation

In data model configuration XML files, you can declare required properties for an entity type during object instantiation. To specify a required entity property is required, set the attribute called `required` to the value `true`. The default is `false`. The required attribute is supported on the `<column>`, `<typekey>`, and `<foreignkey>` definitions of an entity type.

The `required` attribute does not overlap with the behavior of the attribute called `nullok`. The `nullok` attribute affects only the value of that property at commit time, which is the time the object is written to the database. In contrast, the `required` attribute affects only object instantiation. If the required property is an object type, the value must be non-null at the time of object instantiation to prevent Gosu throwing an exception.

Code that uses the `new` operator to instantiate an entity type must include all required properties when constructing an instance of that entity as arguments to the constructor. If the constructor has other constructor arguments, the required properties appear after those other arguments.

For example, suppose an entity called `MyEntity` has one required parameter that is has type `Address`. The expression `new MyEntity()` is a compilation error because the `required` property is missing from the argument list. To fix the error, get a reference to a non-null `Address` entity instance and pass it to the constructor, such as:

```
var e = new MyEntity(myAddress)
```

Optional bundle argument during instantiation

By default, Gosu creates the new entity instance in the current database transaction. Gosu represents the database transaction as a set of entity instance changes encapsulated in a Gosu object called a *bundle*. In nearly all PolicyCenter contexts, this is the best behavior. For example, PolicyCenter correctly rolls back all changes to the database in that transaction if any errors occur before the entire set of changes commit to the database.

Providing a specific bundle during entity instantiation is appropriate only in rare cases in application logic. For example, there are rare cases where there is no current database transaction or an action must be handled asynchronously in a different thread. For these rare cases, pass the bundle as the final argument to the `new` operator expression.

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

Extra database transaction parameter to `new` operator	Meaning	Example
No extra parameter	Create the entity in the current database transaction, which is the current bundle. In almost all cases, use this approach.	`new Note()`
Reference to a bundle	Create the entity instance in the current database transaction indicated by the bundle passed directly to the `new` operator. The bundle must be writable, for example it cannot be a read-only bundle associated with a database query result.	`new Note(myPolicy.Bundle)`
Reference to a Guidewire entity	Create the entity instance in the same database transaction as the entity passed as a parameter. The bundle must be writable, for example it cannot be a read-only bundle associated with a database query result.	`new Note(myPolicy)`

WARNING Be extremely careful if you use bundle and transaction APIs to override the default behavior for bundle management.

Interaction between bundle parameter and required fields

If an entity type has required fields and you need the optional instantiation parameter for the bundle, the constructor argument list must include the bundle parameter before any required properties.

See also

- “Bundles and database transactions” on page 387

Optional omission of type name with the new keyword

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. Some types imported from XSDs have complex and hard-to-read type names. The following simple examples demonstrate the concepts, but do not promote usage of this syntax 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,
// and 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()
```

See also

- “Object initializer syntax” on page 88

Object initializer syntax

Object initializers support setting properties on newly created objects immediately after `new` expressions. Use object initializers for compact and clear object declarations. Object initializers are especially useful in combination 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 braces. A property initializer expression is the following, in order: a colon (`:`), a property name, an equal sign (`=`), and a value or 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 that type.

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() {               // Create a test set.
  :name = "Root",                           // Set the name of the test set.
  :tests = {
    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 that type.
  },
```

```
  :style =
    new TestStyle() {                    // Create and initalize a test style and set the
      :color = Red,                      // style of the test set to that style.
      :number = 5
    }
}
```

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.

See also

- "Basic lists" on page 183
- "Basic hash maps" on page 185

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 use the following operators:

- `>` operator
- `>=` operator
- `<` operator
- `<=` operator

Gosu supports the use of multiple relatational operators to compare multiple values. Add parentheses around each comparison 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, because `false` is equal to `false`, the entire expression evaluates to `true`.

> operator

The `>` operator tests two expressions and returns `true` if the left value is greater than the right value. The operands can be numeric, `String`, or `java.util.Date` data types. The result is `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 and returns `true` if the left value is greater than or equal to the right value. The operands can be numeric, `String`, or `java.util.Date` data types. The result is `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 and returns `true` if the left value is less than the right value. The operands can be numeric, `String`, or `java.util.Date` data types. The result is `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 and returns `true` if the left value is less than or equal to the right value. The operands can be numeric, `String`, or `java.util.Date` data types. The result is `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
- Bit-wise `NOT`
- `Typeof` expressions

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 unary `-` operator before a number to indicate numeric negation. The operand must be a number data type. The result is always the same type as the original number.

Syntax

```
-value
```

Examples

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

Bit-wise NOT

The bit-wise `NOT` operator treats a numeric value as a series of binary digits (bits) and inverts them. This behavior 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
```

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.

```
z = ~7
```

The decimal number 7 is the binary value 0111 with a positive sign bit. By using the bit-wise `NOT`, the expression ~7 evaluates to the decimal value -8. The binary value 0111 reverses to 1000, the binary value for 8, and the sign bit changes as well to produce -8.

Use the bit-wise `NOT` operation to manipulate a bit mask. A *bit mask* is a number or byte field that maintains the state of many items by flags mapping to each bit in the field.

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.

See also

- "Basic type checking" on page 403

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 some programming languages, ternary expressions are known as using the *conditional operator* or *ternary operator*, one which has three operands instead of two.

If your test is for a `null` value, you can use the null-safe default operator to produce even more concise code.

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 the 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
`3 > 4  ? true : false`	`false`
`3*3 == 9 ? true : false`	`true`

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 the result of evaluating the conditional expression. The true and false expressions often, but not always, evaluate to the same type.

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`. 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 expressions in a ternary expression be compatible so that Gosu can reconcile their types. If the expressions 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 acestor 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 depends on the current state of the system. Different type checking keywords produce different results. For example, if the contact in the following example is new, `ternaryResult` is of type `String`.

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

print(ternaryResult typeis Object)
print(ternaryResult typeis String)
print(ternaryResult typeis Boolean)
print(typeof ternaryResult)
```

The preceding example produces the following output.

```
true
true
false
String
```

If the contact in the following example is not new, `ternaryResult` is of type `Boolean`.

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

The preceding example produces the following output.

```
true
false
true
Boolean
```

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 to one of the expressions `b`, `d`, `f`, or `g`.

See also

- “Logical expressions” on page 83
- “Null-safe default operator” on page 98

Special Gosu expressions

Gosu syntax supports accessing functions, static methods, static property paths, entity literals, and typekey literals.

Function calls

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

Syntax

```
functionName(argumentList)
```

Examples

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

Static method calls

This expression calls a a static method on a type with an optional list of arguments and returns the result.

Syntax

```
typeExpression.staticMethodName(argumentList)
```

Examples

Expression	Result
`java.lang.System.currentTimeMillis()`	Current time
`java.util.Calendar.getInstance()`	Java Calendar

See also

- “Modifiers” on page 157

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
`Claim.TypeInfo`	`Claim typeInfo`
`LossCause.TC_HAIL`	`"hail" typekey`

Expression	Result
`java.util.Calendar.FRIDAY`	`Friday value`

See also

- "Modifiers" on page 157

Entity and typekey type literals

Gosu supports referencing an `entity` or `typekey` type by relative name, or by fully qualified name:

- 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

Many operations in Gosu are null-safe. These operations do not throw a null pointer exception (NPE) at run time if they encounter a `null` value. Gosu provides a set of null-safe operators to protect other operations from these exceptions.

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 use this operator if you want a null-safe method invocation.

See also

- "Property assignment triggering instantiation of intermediate objects" on page 56
- "Null-safe method access" on page 98

How the standard period operator handles null

The standard period (`.`) 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`. A `null` value at any location

in an object path short-circuits evaluation and results in a `null` value, with no exception being thrown. This feature is called null-safe short circuiting 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 a `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
```

By using null-safe paths, Gosu 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 descendents of `Object`) can never contain the value `null`. Thus, Gosu cannot return `null` from that expression, and any casting from `null` to the primitive type would be meaningless. Therefore, Gosu throws a null pointer exception in those conditions.

WARNING In Studio, Gosu code in the Gosu Scratchpad has different compiler behavior than any other part of Studio. If you run code in Gosu Scratchpad directly in Studio without being connected to a running server, the code compiles and runs locally in Studio with different behavior. Most notably, the behavior of null safety in the period operator is different. In nearly all other contexts in Studio, the period operator is null safe. However, in Gosu Scratchpad in Studio without a connection to a running server, Gosu is not null-safe. Code that might not throw an exception in a Gosu class might throw an exception in Gosu Scratchpad. If Studio is connected to a running server, Studio sends the Gosu code to the server. The PolicyCenter server uses the standard null-safe behavior for the period operator.

See also

- “Property assignment triggering instantiation of intermediate objects” on page 56

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.

See also

- “Property assignment triggering instantiation of intermediate objects” on page 56

Null-safe method access

By default, method calls are not null-safe. If the right side of a period character is a method call, Gosu throws a null-pointer exception (NPE) if the value on the left side of the period is `null`.

A null-safe method call does not throw an exception if the left side of the period character evaluates to `null`. Gosu just returns `null` from that expression. Using the `?.` operator calls the method with `null` safety. This operator behaves in the same way as the null-safe period operator for properties.

See also

- “How the standard period operator handles null” on page 96

Null-safe default operator

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

You can chain the null-safe default operator to get the first non-null value in a set of items. If every value in the chain is `null`, the result of the chain is `null`. For example, the following code gets the first day of the week that has an assigned task or returns `null` if no day has a task:

```
var firstTask = dayTask[0] ?: dayTask[1] ?: dayTask[2] ?: dayTask[3] ?: dayTask[4]
```

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 typing just 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 evaluates the index subexpression and indexes the array, list, or map. Finally, Gosu returns the result, just as for the typical use of the brackets for indexing lists, arrays, and maps.

Null-safe indexing does not protect against array-index out-of-bounds exceptions. If the array is non-null, but the index is greater than or equal to the size of the array, Gosu throws an out-of-bounds exception at run time. Any value of the index causes this exception if the array is empty rather than `null`. To test for an empty array, use the `HasElements` property.

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 `?%`.

Handling null values in logical expressions

If your data can provide `null` values, you need to be aware of how Gosu handles these values in logical expressions. Using the null-safe operators can propagate `null` values from their original sources into other objects and variables in your code.

- In an equality test, Gosu returns `true` if both sides of the operator are `null` and `false` otherwise.
- In a Boolean expression that uses the `and`, `or`, or `not` operators, Gosu treats `null` as a value of `false`.

Be particularly aware of the possibility of `null` values in code that uses the `and`, `or`, or `not` operators. If you use null-safe method operators in logical expressions, your code can produce unexpected results:

```
var nullStr : String // A string that contains the null value

// Test whether the string is empty
if (nullStr?.isEmpty()) {
  print("Empty string") // This message does not appear even though there is nothing in the string.
  // Code here to put a value into an empty string does not execute for the null string.
```

```
}
// Test whether the string is not empty
if (!nullStr?.isEmpty()) {
  print("Got here!") // This message does appear, but the string has no contents.
  // If you put code here to use the string value, you risk a null-pointer exception.
}
```

See also

- "How null values get in the database" on page 325

Best practices for using null-safe operators

Be aware of the consequences of using the null-safe operators. If the null-safe expression returns a value of `null`, your subsequent code must be able to handle that value. If the argument on the left of a null-safe operator is `null`, the expression returns a value of `null`, or its equivalent for a primitive type. For example, using a null-safe operator in an expression that returns an `int` value produces a value of 0 if the argument on the left of the operator is `null`.

Do not use null-safe operators merely to avoid any null-pointer exceptions. Using these operators is a signal to other code developers that the code path can accept and return `null` values. Use the null-safe operators only if your code does not need to perform additional actions for an object path expression that evaluates to `null`.

Do not use null-safe operators if you know that an object is never going to be `null`. For example, if you create a new object, subsequent code has no need to use null-safe operators on that object.

```
uses gw.api.util.DateUtil

var startTime = DateUtil.currentDate()
// No need to use null-safe operators to call methods on startTime
var terminationTime = startTime.addHours(2) // CORRECT
var halfTime = startTime?.addHours(1) // DO NOT USE THIS FORM. startTime is never null.
```

Similarly, if a function in a class uses the `this` name to reference the current object, do not use the null-safe operators, such as `this?.methodName()`. The `this` object never has a value of `null`.

List and array expansion expressions (*.)

Gosu includes a special operator for array expansion and list expansion. The expansion operator is an asterisk followed by a period. For example, the following line of code gets the length property of every item in an array called `names`:

```
names*.length
```

See also

- "List expansion (*.)" on page 187

chapter 4

Statements

This topic describes important concepts in writing more complex Gosu code to perform operations required by your business logic.

Gosu statements

A Gosu expression has a value, but Gosu statements do not. The result of a Gosu expression can be passed as an argument to a function. A Gosu statement does not have a result that can be passed as an argument to a function.

For example, the following lines 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 lines 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 function arguments.

See also

- “Blocks” on page 175

Statement lists

A statement list is a list containing zero or more Gosu statements enclosed by braces ({}).

The Gosu standard is always to omit semicolon characters at the ends of lines of Gosu code. Code is more readable without optional semicolons. In the rare cases in which you type multiple statement lists on one line, such as within block definitions, do use semicolons to separate statements.

Syntax

```
{ statement-list }
```

Multi-line example (no semicolons)

```
{
  var x = 0
  var y = myfunction( x )

  print( y )
}
```

Single-line example (semicolons)

```
var adder = \ x : int, y : int -> { print("I added!"); return x + y; }
```

See also

- "General coding guidelines" on page 465

Using the operator new as a statement

Use the `new` operator to instantiate an object. Although it is often used as an expression, `new` can also be a statement. For some types, this functionality 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 these 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 merely create the book object and pass the bookshelf as a constructor argument:

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

See also

- "Using the operator new in object expressions" on page 86

Gosu variables

To create and assign variables, consider the type of the variable as well as its value.

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 value assigned to it.

Syntax

```
var identifier [ : type-literal ] = expression
var identifier : type-literal [ = expression ]
```

Examples

```
var age = 42
var age2 : int
var age3 : int = "42"
```

```
var c : Claim
...
```

Variable assignment

Gosu uses the standard programming assignment operator = to assign the value on the right-side of the statement to the item on the left-side of the statement.

Syntax

```
variable = expression
```

Examples

```
count = 0
time = now()
```

Gosu also supports compound assignment operators that perform an action and assign a value in one action. The following lists each compound operator and its behavior. The examples assume the variables are previous declared as `int` values.

Operator	Description	Examples
=	Simple assignment to the variable on the left-hand side of the operator with the value on the right-hand side.	`i = 10` Assigns value 10.

+=	Increases the value of the variable by the amount on the right-hand side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.	`i = 10`
		`i += 3`
		Assigns value 13.

-=	Increases the value of the variable by the amount on the right-hand side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.	`i = 10`
		`i -= 3`
		Assigns value 7.

*=	Multiplies the value of the variable by the amount on the right-hand side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.	`i = 10`
		`i *= 3`
		Assigns value 30.

/=	Divides the value of the variable by the amount on the right-hand side of the operator.	`i = 10`
		`i /= 3`
		Assigns value 3.
		For the `int` type, there is no fraction. If you used a floating-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.	`var i = 10`
		`i %= 3`
		Assigns value 1.
		This value 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`.

Operator	Description	Examples

&&=	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`.

&=	Performs a bitwise AND operation with the original value of the variable and value on the right-hand side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.	`i = 10`
		`i &= 15`
		Assigns value 10.
		The decimal number 10 is 1010 binary. The decimal number 15 is 1111 binary. This code does a bitwise AND between value 1010 and 1111. The result is binary 1010, which is decimal 10.
		Contrast with this example:
		`i = 10`
		`i &= 13`
		Assigns value 8.
		The decimal number 10 is 1010 binary. The decimal number 13 is 1101 binary. This does a bitwise AND between value 1010 and 1101. The result is binary 1000, which is decimal 8.

^=	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.	`i = 10`
		`i ^= 15`
		Assigns value 5.
		The decimal number 10 is 1010 binary. The decimal number 15 is 1111 binary. This code does a bitwise exclusive OR with value 1010 and 1111. The result is binary 0101, which is decimal 5.
		Contrast with this example:
		`i = 10`
		`i ^= 13`
		Assigns value 7.
		The decimal number 10 is 1010 binary. The decimal number 13 is 1101 binary. This does a bitwise AND between value 1010 and 1101. The result is binary 0111, which is decimal 7.

|=	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.	`i = 10`
		`i |= 15`
		Assigns value 15.
		The decimal number 10 is 1010 binary. The decimal number 15 is 1111 binary. This code does a bitwise inclusive OR with value 1010 and 1111. The result is binary 1111, which is decimal 15.
		Contrast with this example:
		`i = 10`
		`i |= 3`
		Assigns value 11.
		The decimal number 10 is 1010 binary. The decimal number 13 is 1101 binary. This does a bitwise AND between value 1010 and 1101. The result is binary 0111, which is decimal 11.

Operator	Description	Examples

<<=	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.	`i = 10`
		`i <<= 1`
		Assigns value 20.
		The decimal number 10 is 01010 binary. This code does a bitwise left shift of 01010 one bit to the left. The result is binary 10100, which is decimal 20.
		Contrast with this example:
		`i = 10`
		`i <<= 2`
		Assigns value 40.
		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.

>>=	Performs a bitwise right shift with the original value of the variable and value on the right side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.
	IMPORTANT: for signed values, this operator automatically sets the high-order bit with its previous value for each shift. This preserves the sign (positive or negative) of the result. For signed integer values, this is the usually the appropriate behavior. Contrast this operatorwith the >>>= operator.
		`i = 10`
		`i >>= 1`
		Assigns value 5.
		The decimal number 10 is 1010 binary. This code does a bitwise right shift of 1010 one bit to the right. The result is binary 0101, which is decimal 5.
		Contrast with this example:
		`i = -10`
		`i >>= 2`
		Assigns value -3.
		The decimal number -10 is 11111111 11111111 11111111 11110110 binary. This code does a bitwise right shift two bits to the right, filling in the top sign bit with the 1 because the original number was negative. The result is binary 11111111 11111111 11111111 11111101, which is decimal -3.

>>>=	Performs a bitwise right shift with the original value of the variable and value on the right side of the operator. Next, Gosu assigns this result to the variable on the left-hand side.
	IMPORTANT: this operator sets the high-order bit with its previous value for each shift to zero. For unsigned integer values, this is the usually the appropriate behavior. Contrast this operator with the >>= operator.
		`i = 10`
		`i >>>= 1`
		Assigns value 5.
		The decimal number 10 is 1010 binary. This code does a bitwise right shift of 1010 one bit to the right. The result is binary 0101, which is decimal 5.
		Contrast with this example:
		`i = -10`
		`i >>>= 2`
		Assigns value 1073741821.
		The negative decimal number -10 is 11111111 11111111 11111111 11110110 binary. This code does a bitwise right shift two bits to the right, with no filling of the top bit. The result is binary 00111111 11111111 11111111 11111101, which is decimal 1073741821. The original was a negative number, but in this operator that bit value is filled with zeros for each shift.

++ unary operator	Adds one to the current value of a variable. Also known as the increment-by-one operator. The unary ++ and -- operators must always appear after the variable name.	`i = 10`
		`i++`
		Assigns value 11.

-- unary operator	Subtracts one from the current value of a variable. Also known as the decrement-by-one operator. The unary ++ and -- operators must always appear after the variable name.	`i = 10`
		`i--`
		Assigns value 9.

Compound assignment compared to expressions

The table above lists a variety of compound assignment operators, such as `++`, `--`, and `+=`.

These operators form statements, rather than expressions.

The following Gosu is valid

```
while(i < 10) {
  i++
  print( i )
}
```

The following Gosu is invalid because statements are not permissible 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.

See also

- “Variable assignment” on page 103

Gosu conditional execution and looping

Gosu uses the multiple constructions to perform program flow.

if - else statements

The most commonly used statement block within the Gosu language is the `if` block. The `if` block uses a multi-part construction. The `else` block is optional.

Syntax

```
if (<expression>) <statement>
[ else <statement> ]
```

Example

```
if (a == b) { print("a equals b") }

if (a == b || b == c) { print("a equals b or b equals c") }
else { print("a does not equal b and b does not equal c") }

if (a == b) { print("a equals b") }
else if (a == c) { print("a equals c") }
else { print("a does not equal b, nor does it equal c") }
```

To improve the readability of your Gosu code, Gosu automatically downcasts after a `typeis` expression if the type is a subtype of the original type. This is particularly 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.

See also

- "Basic type checking" on page 403

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>` 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.

If the expression evaluates at run time to `null`, the body of the loop block is skipped entirely.

Gosu provides backwards compatibility for the use of an older Gosu style `foreach` statement. Better style is to use the `for` statement instead.

See also

- "Intervals" on page 131
- "Array list access with array index notation" on page 68

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 a 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 code prints the index of the highest score in an array of test scores.
// This particular example prints "3".

var testScores = new int[] {91, 75, 97, 100, 89, 99}
print(getIndexOfHighestScore(testScores))

function getIndexOfHighestScore(scores : int[]) : int {
  var highIndex = 0

  for (score in scores index i) {
    if (score > scores[highIndex]) { highIndex = i }
  }

  return highIndex
}
```

The result of running this code is the following line:

```
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)
}
```

The results of running this code are the following lines:

```
red
green
blue
```

Examples of Gosu conditional iteration and looping

The following examples illustrate the different techniques 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, which must evaluate to `true` or `false`, and uses the Boolean result 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 is that Gosu tests the expression for validity after executing the statement block, instead of prior to executing the statement block. The statements in the statement block execute at least once when execution first accesses the block.

- 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 evaluates the expression, and then iterates through the case expressions in order until it finds a match.

- If a case value equals the expression, Gosu executes its accompanying statement list. Statement execution continues until Gosu encounters a `break` statement, or the switch statement ends. Gosu continues to the next case and 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. In most cases, you implement a default case to handle any unexpected conditions.

Syntax

```
switch (<expression>) {
  case label1 :
    [statementlist1]
    [break]
  [ ...
  [ case labelN :
    [statementlistN]
    [break] ] ]
  [ default :
    [statementlistDefault]]
}
```

Example

```
switch (strDigitName) {
  case "one":
    strOrdinalName = "first"
    break
  case "two":
    strOrdinalName = "second"
    break
  case "three":
    strOrdinalName = "third"
    break
  case "five":
    strOrdinalName = "fifth"
    break
  case "eight":
    strOrdinalName = "eighth"
    break
```

```
  case "nine":
    strOrdinalName = "ninth"
    break
  default:
    strOrdinalName = strDigitName + "th"
}
```

To improve the readability of your Gosu code, Gosu automatically downcasts the object after a `typeis` expression if the type is a subtype of the original type. This is particularly valuable for `if` statements and similar Gosu structures such as `switch`. Within the Gosu code bounded by the `if` or `switch` statement, you do not need to do casting (`as` *`TYPE`* expressions) to that subtype for that case. Because Gosu confirms that the object has the more specific subtype, Gosu implicitly considers that variable's type to be the subtype for that block of code. There are several special cases that turn off the downcasting.

See also

- "Basic type checking" on page 403

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.

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 : int ) : int {
  return n * n
}

// Compile error "Cannot return a value from a void function."
private function myfunction() {
  return "test for null value"
}

function fibonacci( n : int ) : int {
  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
}
```

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.

See also

- "What are blocks?" on page 175
- "Modifiers" on page 157

Named arguments and argument defaults

Gosu provides some optional features that make function calls very readable.

Calling named arguments

In your 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)
```

If you pass series of one or more comma-separated arguments, it is difficult or impossible to tell what each value represents. It can be difficult to visually confirm the argument order in code that was written long ago or written by others.

Alternatively, for each argument you can explicitly specify the argument name. Instead of just the argument value, pass the following items in this order:

1. A colon character
2. The argument name
3. The equal sign
4. The value

For example:

```
someMethod(:redisplay=true, :sendUpdate=false)
```

Defining argument defaults

You can optionally provide default argument values in function declarations. By providing a default argument value, you permit a function caller optionally to omit that argument. To declare a default, follow the argument name with an equal sign and then the value. If the function caller passes the argument, the passed-in value overrides any declared default value.

For example, consider the following function that prints `String` values 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
    }
  }
}
```

In the `printWithPrefix` declaration, the prefix value has the default value `" ---> "`. To use the default values, call the `printWithPrefix` method and omit the optional arguments.

The following example calls the printWithPrefix method using the default value:

```
var c = new MyClass({"hello", "there"})

// Because the argument has a default, it is optional -- you can omit it
c.printWithPrefix()
```

The following example calls the printWithPrefix method providing an explicit value:

```
var c = new MyClass({"hello", "there"})

// Specify the parameter explicitly (ignores default) as traditional unnamed argument
c.printWithPrefix(" next string is:")

// Specify the parameter explicitly (ignores default) as named argument
c.printWithPrefix(:prefix= " next string is:")
```

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

Interaction of named arguments and argument defaults

When you call a function with a multiple arguments, you can explicitly name some of the arguments and not others. The following are the rules for the calling convention of functions using the named argument convention and argument defaults:

- The named arguments can appear in any order.
- For any non-named arguments in the function call, the order 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.

See also

- “Modifiers” on page 157

Importing types and package namespaces

To use types and namespaces in Gosu scripts without fully qualifying the full class name including the package, use a Gosu `uses` statement. The `uses` statement behaves similarly to the Java language’s `import` statement. To indicate that you are importing static members of a class, you use a `#` character and the members to import after the type name.

Always put `uses` statements at the beginning of the file or script. Do not add these lines within a Gosu method declaration.

Syntax

After the `uses` statement, specify a package namespace or a specific type such as a fully qualified class name.

```
uses type
uses namespace
uses type#static member
uses type#all static members
```

Namespaces can be specified with an asterisk (*) character to indicate a hierarchy. For example:

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

Example 1

The following code uses a fully qualified type name.

```
var myInputStream = new java.io.FileInputStream()
```

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

```
// This "uses" expression provides access to the java.io.FileInputStream class
uses java.io.FileInputStream

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

Example 2

The following code uses a fully qualified type name.

```
var myInputStream = new java.io.FileInputStream()
```

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

```
// This "uses" expression provides access to all the classes in the java.io package hierarchy
uses java.io.*

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

Example 3

The following code uses a qualified type name. Because classes in the `java.lang` package are always in scope, you do not need to specify `java.lang`.

```
var circleRadius = 2.0
var circleCircumference = circleRadius * circleRadius * Math.PI
```

Instead, you can use the following code that declares a single static field from the `java.lang.Math` class.

```
// This "uses" expression provides access to the PI constant in the java.lang.Math class
uses java.lang.Math#PI

var circleRadius = 2.0
// Use this simpler expression without specifying the class name:
var circleCircumference = circleRadius * circleRadius * PI
```

Example 4

The following code uses a qualified type name. Because classes in the `java.lang` package are always in scope, you do not need to specify `java.lang`.

```
var mySquareArea = 1000
var mySideLength = Math.sqrt(mySquareArea)
```

You can use any of the angle constants in your code.

```
// This "uses" expression provides access to all the static members in the java.lang.Math class
uses java.lang.Math#*

var mySquareArea = 1000
// Use this simpler expression without specifying the class name:
var mySideLength = sqrt(mySquareArea)
```

Packages always in scope

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

- `gw.lang.*`
- `java.lang.*`
- `java.util.*`
- `dynamic.*`
- The default, empty package
- `entity.*`
- `typekey.*`
- `pcf.*` – only for expressions inside PCF files.
- `productmodel.*`

List

The Gosu type system performs special treatment for the type `List`. Gosu resolves `List` to `java.util.List` in general use but to `java.util.ArrayList` if code creates an untyped list. For example, the following code creates an `ArrayList`:

```
var x = {}
```

Importing static members

The Gosu `uses` statement supports importing static features from a type so you can use them later in your code without verbose syntax. Static features are members of a type that are defined with the `static` keyword, and so appear on the type itself rather than an instance of the type. The Gosu static imports feature is similar to the Java equivalent, but with more concise syntax that does not require the `static` keyword during import.

Static features that you can import include:

- Static properties
- Static fields
- Static methods

Gosu static imports are particularly useful to support the Gosu binding expressions feature.

To import static features, at the top of the file, write a `uses` line that specifies a type, followed by the `#` character, than the static member name. For the static member name, list the property name, the field name, or a method signature with type names only (omit the argument names). The use of the `#` character in this case reflects the Gosu feature syntax for a property or method on a type. This syntax is sometimes called *feature literal* syntax.

To import all static features for a type, use the `*` character instead of feature literal syntax for a specific member.

The following example imports one method:

```
uses org.junit.Assert#assertEquals(String, Object, Object)
```

The following example imports all static features from a class

```
uses gw.util.money.IMoneyConstants#*   // Imports all static features from IMoneyConstants
```

> **Note:** Use static imports with caution. Excessive use of unqualified names of static members from outside the scope of a class creates code that is confusing to read.

Example usage of static imports

Create the following example class `example.TestClass`.

```
package example
```

```
class TestClass {

  // Static field (a field directly on the type, not an instance of it)
  static public var OneHundred : Integer = 100

  // Static method (a method directly on the type, not an instance of it)
  static function helloWorld() {
    print("Hello world")
  }
}
```

Note that the class contains a static field and a static method. Without using static imports, Gosu code in other classes needs to qualify the field or method:

```
uses example.TestClass

print(2 + TestClass.OneHundred)

TestClass.helloWorld()
```

If you import the static members from the type, you can omit the type name before using static members:

```
// Import all static members from a type (note the #* at the end)
uses example.TestClass#*

// You can now use a static field without specifying what type it is on
print(2 + OneHundred)

// You can call a static function without specifying what type it is on
helloWorld()
```

chapter 5

Exception handling

Gosu supports the following standard exception handling constructions from other languages such as throw statements, `try/catch/finally` blocks, and special Gosu statements such as the `using` keyword.

Handling exceptions with try/catch/finally

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 run time. If errors occur that the script does not handle, Gosu provides its normal error message, as if no error handling existed.

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 to handle the error.
- In the catch block, Gosu passes as an argument to catch block a value that represents the exception that occurred in the `try` block. If an error is thrown from Java code, the value that the catch block gets is the exception thrown. For exceptions directly thrown from Gosu code, any value that is not a subclass of `RuntimeException` is wrapped in a `RuntimeException` object.
- 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 Gosu runs all statements in the `try` block or after running the error handling code in the `catch` block, Gosu runs the `finally` block.

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. It is important to understand that , Gosu always runs the `finally` block.

In a `finally` block, Gosu does not permit the `return`, `break`, or `continue` statements.

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 gw.lang.parser.EvaluationException: an error
Nested FINALLY running...
Outer CATCH caught gw.lang.parser.EvaluationException: gw.lang.parser.EvaluationException: an error
        rethrown
Outer FINALLY running
```

See also

- “Throwing exceptions” on page 121
- “Catching exceptions” on page 120

Catching exceptions

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

The standard syntax for `catch` is the following, which catches all exceptions by specifying the class `Exception`:

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

To catch a single specific exception type, specify a subclass such as `IOException` instead of `Exception`. The technique of catching a named exception is called *checked exceptions*. The recommended Gosu coding style is not to use checked exceptions, although this technique is supported.

The following code is an example of checked exceptions:

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

Add a `finally` block at the end to perform cleanup code that runs for both error and 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. Best practice is to catch `Exception`, not `Throwable`. Use `catch(e : Exception)` not `catch(e : Throwable)`. Catching `Throwable` can catch serious infrastructure problems like `OutOfMemoryException` or `AssertionFailedException`. Typical code

cannot handle those types of exceptions, so catch `Exception` so that these serious infrastructure exceptions propagate upward.

Throwing exceptions

The `throw` statement generates an error condition that 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 these statements only for handling actual error conditions.

The following example throws 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`, which has a `Message` property that contains the `String` data. Your error-handling code can use the `Message` property for logging or displaying messages to users.

You could rewrite the previous `throw` code example as the concise code:

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

Syntax

```
throw <expression>
```

In the following example, notice how the error message changes if the value of `x` changes from 0 to 1.

Example

```
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
```

Object life-cycle management with using clauses

If you have an object with a complete life cycle in a particular 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 simplified.

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 by 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.

Several categories of objects 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 compiler error.

If Gosu detects that an object is in more than one category, at run time, Gosu considers the object to be in only one category. Gosu selects the category by the following precedence: disposable, closeable, reentrant. For example, if an object has a `dispose` and `close` method, Gosu only calls the `dispose` method.

Use the `return` statement to return values from `using` clauses.

If Gosu successfully evaluates and initializes all variables, each variable is examined in order. For each object, if the object has an enter action, that action is taken:

- If the object implements the `Lock` interface, Gosu calls `Lock.lock()`
- If the object implements the `IReentrant` interface, Gosu calls `IRenntrant.enter()`
- If the object has a a method named `lock` and it takes no parameters, Gosu calls the `lock` method.
- If the object was cast to the `IMonitorLock` interface, Gosu synchronizes on the variable

If the enter action completes without an exception, Gosu guarantees that the corresponding exit action for that object will run. If no enter action applies to the object, Gosu guarantees that the exit action for the object will run. Exit actions are called in the reverse order of declaration in the `using` clause. Exit actions are as follows:

- If the object implements the `IDisposable` interface, Gosu calls `IDisposable.dispose()`
- If the object has a method named `dispose` and it takes no parameters, Gosu calls the method.
- If the object implements the `Closeable` interface, Gosu calls `Closeable.close()`
- If the object has a method named `close` and it takes no parameters, Gosu calls the method.
- If the object implements the `IReentrant` interface, Gosu calls `IRenntrant.exit()`
- If the object implements the `Lock` interface, Gosu calls `Lock.unlock()`
- If the object has a method named `unlock` that takes no parameters, Gosu calls the `unlock` method.
- If the object was cast to the `IMonitorLock` interface, Gosu unsynchronizes on the variable

If an exception occurs during the execution of an exit action of an object, the exit actions defined before that exit action run, and then the exception throws. If an exception occurs in more than one exit action, the outermost exception, which occurs later in the code, takes precedence and is thrown to the surrounding code.

Assigning variables inside a using expression declaration

The `using` clause supports assigning a variable inside the declaration of the `using` clause. This feature is useful if you want to reference the expression that you pass to the `using` expression from inside the `using` clause.

For example, suppose you call a method that returns a file handle and you pass that handle to the `using` clause as the lock. From within the `using` clause contents, you probably want to access the file so that you can iterate across its contents.

To simplify this kind of code, use the `var` keyword to assign a variable to the expression:

```
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 a 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. You cannot pass more than one reentrant object, which is an object that implements `IReentrant` or `IMonitorLock` interfaces or has a `lock` method.

For example:

```
using ( _file1, _file2, _file3) {
  // Do your main work here
}
```

To pass multiple reentrant objects, nest multiple `using` clauses, such as:

```
using ( lock1 ) {
  using ( lock2 ) {
```

```
  ...
  }
}
```

You can combine the multiple item feature with the ability to assign variables. 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, Gosu performs the following actions:

1. Gosu skips initialization for any subsequent items not yet initialized, in other words the items 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.

See also

- “Assigning variables inside a using expression declaration” on page 123

Disposable objects

Disposable objects are objects of which 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 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 throws an exception.

The following example shows a basic disposable object:

```
// Create a simple disposable class that 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.
```

A `using` clause supports passing multiple objects. If you pass multiple objects to the `using` clause, exit actions (closing, disposing, unlocking) happen in reverse order of the object declarations.

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.Closeable`, which contains only a single method called `close`. This method takes no arguments. Use a type that implements `Closeable` if possible due to faster run time performance.
- Has a `close` method with no arguments, even if it does not implement the `Closeable` 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 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 variables for the `using` clause:

```
using (var reader = new FileReader("c:\\temp\\usingfun.txt"),
       var writer = new FileWriter("c:\\temp\\usingfun2.txt")) {
  writer.write( StreamUtil.getContent( reader ) )
}
```

A `using` clause supports passing multiple objects. If you pass multiple objects to the `using` clause, exit actions (closing, disposing, unlocking) happen in reverse order of the object declarations.

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

A *reentrant* object supports multiple concurrent uses of a single lock in a single thread. Reentrancy maintains a count of the number of times the thread sets and releases the lock. In reentrant lock use, the thread does not deadlock on setting an already locked lock, nor release the lock when other code in the thread still retains that lock. Reentrant objects help manage safe access to data that is shared by reentrant 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 at least one of the following attributes:

- The object implements the `java.util.concurrent.locks.Lock` interface. For example, the following Java classes in that package implement this interface: `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.
- The object implements the Gosu class `gw.lang.IReentrant`. This interface contains two methods with no arguments: `enter` and `exit`. Your implementation 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 that use locks by implementing `java.util.concurrent.locks.Lock`, a `using` clause simplifies the code. The `using` statement always cleans up the lock, even if the code throws an exception.

```
uses java.util.concurrent

// 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 code uses the object itself, by using the keyword `this`, as the monitor lock:

```
function monitorLock() {
  using ( this as IMonitorLock ) {
    // Do stuff
  }
}
```

This approach is equivalent to a `synchronized` block in the Java language.

The following code uses the `java.util.concurrent.locks.ReentrantLock` class using more verbose `try` and `finally` statements. This approach is not recommended:

```
// In your class variable definitions...
var _lock : ReentrantLock = new ReentrantLock()

function useReentrantLockOld() {
  _lock.lock()
  try {
    // Do your main work here
  }
  finally {
    lock.unlock()
  }
}
```

Alternatively, you can do your change in a Gosu block. This approach is not recommended. Returning the value from a block imposes more restrictions on how you implement `return` statements. It is typically better to use the `using` statement structure shown earlier in this topic.

```
uses java.util.concurrent

...

property get SomeProp() : Object {
  var retValue : Object
  _lock.with( \-> {
    retValue = _someVar.someMethod()
  })
  return retValue
}
```

A `using` clause supports passing multiple objects. If you pass multiple objects to the `using` clause, exit actions (closing, disposing, unlocking) happen in reverse order of the object declarations.

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.

See also

- `http://en.wikipedia.org/wiki/Monitor_(synchronization)`
- “Object life-cycle management with using clauses” on page 122
- “Concurrency” on page 417
- “Using profiler tags” on page 127

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 )
}
```

See also

- *System Administration Guide*

Returning values from using clauses

You can return values from `using` clauses by using the standard `return` statement. If you return a value from a `using` clause, Gosu considers the clause complete, and therefore calls your object's final life-cycle 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.

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
}
```

See also

- "Disposable objects" on page 124
- "Closeable objects and using clauses" on page 125
- "Reentrant objects and using clauses" on page 126

Optional use of a finally clause with a using clause

In some cases, you 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: In 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: In the finally clause.")
  print("value of d.OpenFlag = ${d.OpenFlag}")
}
print("LOG: Outside the 'using' clause.")
```

This example prints:

```
LOG: Created my object!
LOG: In the body of the 'using' statement.
value of d.OpenFlag = true
LOG: Disposed of my object! Note that you must support multiple calls to dispose.
LOG: In the finally clause.
value of d.OpenFlag = false

java.lang.ArithmeticException: / by zero
[...]
```

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 provides 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
```

- *`expressionBoolean`* is an 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.

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

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

chapter 6

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 set is `0..5`. Intervals are useful for concise, readable `for` loops. Intervals can be of many types including numbers, dates, dimensions, and names. You can add custom interval types. In some programming languages, intervals are 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 set of integers is `0..10`. Intervals are particularly useful for concise, readable `for` loops.

For example, consider this concise, readable code:

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

This code 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 code 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 of many 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 previous `0..10` example. 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. You can then use your new intervals in `for` loop declarations.

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

Omitting an initial or ending value

In the simplest case, a Gosu interval iterates from the beginning endpoint value to the ending endpoint value, including 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. In other cases, you want to exclude the endpoint value from the interval.

To exclude an endpoint in Gosu, type the pipe `"|"` character:

- To omit the value of the starting endpoint, type the pipe character after the starting endpoint.
- To omit the value of the ending endpoint, 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.

See also

- "Writing your own interval type" on page 133

Reversing interval order

Sometimes you want a loop to iterate across elements in an interval in reverse order. To do this iteration, reverse the values of the beginning endpoint and ending endpoint for 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, these intervals are the same objects but Gosu marks `5..0` as being in reverse order.

For example, this code 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 the interval as reversed. Use the following syntax:

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

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 your definition of the interval.

Granularity of interval steps and units

You can customize the granularity with the `step` and `unit` builder-style methods on an interval. You use the `step` method to set the number of items to step or skip by. The `unit` method specifies the unit, which is 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, 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, which is an enumeration constant. You do not need to qualify `WEEKS` 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 of either of the two basic types of intervals:

- *Iterable intervals*, which are intervals that you can iterate across, such as in `for` loop declarations
- Non-iterable intervals

For typical code, iterable intervals are more useful because they can simplify common coding patterns in loops.

In some circumstances, you need 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.

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` Gosu interface, which 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` Gosu class and 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.pl.gosu.interval

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

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 a class knows how to find the next and previous items in the sequence. Sequenceable and interval types have a lot in common. Both types 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 me thods and declare your class to implement this interface.

```
public interface ISequenceable<E extends ISequenceable<E, S, U>, S, U> extends Comparable<E> {
  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 );
}
```

Method	Parameter	Description
`nextInSequence`		Returns `E` *`element`*, the next element in the sequence
	`S` *`step`*	An alias for the column name in the row query result
	`U` *`unit`*	A path expression that references the column to include in the row query result. For example, you can use a feature literal expression.
`nextNthInSequence`		Returns `E` *`element`*, the the element that is `int` *`index`* next in the sequence
	`S` *`step`*	An alias for the column name in the row query result
	`U` *`unit`*	
	`int` *`index`*	
`previousInSequence`		Returns `E` *`element`*, the previous element in the sequence
	`S` *`step`*	An alias for the column name in the row query result
	`U` *`unit`*	
`previousNthInSequence`		Returns `E` *`element`*, the element that is `int` *`index`* previous in the sequence
	`S` *`step`*	An alias for the column name in the row query result
	`U` *`unit`*	
	`int` *`index`*	

The syntax for the interface might look unusual because of the use of Gosu generics. This syntax indicates that the interface 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>
```

This *bidirectional* interface can fetch both next and previous elements. Gosu needs this capability to handle navigation from either endpoint in an interval to support the reverse mode. Gosu also requires that the class know how to jump to an element by its index in the series. This functionality 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.gosu.interval

enum ColorSequenceable
        implements gw.lang.reflect.interval.ISequenceable<ColorSequenceable, 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 ): ColorSequenceable {
    return ColorSequenceable.AllValues[this.Ordinal + stp]
  }

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

  override function previousInSequence( stp : Integer, unit : java.lang.Void ) : ColorSequenceable {
    return ColorSequenceable.AllValues[this.Ordinal - stp]
  }

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

To 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(
        ColorSequenceable.Red, ColorSequenceable.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(
        ColorSequenceable.Red, ColorSequenceable.Blue, 1, null, false, false, false )

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

This code prints:

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

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

See also

- “Enumerations” on page 169
- “Generics” on page 221

Custom iterable intervals using manually written iterators

If your items are not sequenceable, you can make an iterable interval class for those items, but you write more code to implement all necessary methods.

Create a custom iterable interval using manually written iterator classes

You perform the following general tasks to write a manual iterator class.

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, which is an `Integer`.
   - 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 the interval does not have units, use `java.lang.Void` for this dimension of the parameterization. Make sure that you use the capitalization of this type. In Gosu, as in Java, `java.lang.Void` is the special type of the value `null`.
   - The type of your custom interval. This type is self-referential because some of the methods return an instance of the interval type itself.

   The documentation provides an example of a class that extends the type:

   ```
   IterableInterval<Color, Integer, void, ColorInterval>
   ```
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. A straightforward way to implement these methods is to define iterator classes as inner classes to your main class.

   Your class must be able to return two types of iterators, one iterating forward, and one iterating in reverse. 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.

See also

- “Inner classes” on page 164
- “Generics” on page 221
- “Example: Color interval written with manual iterators” on page 136

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 Gosu shortcuts for custom iterable intervals using sequenceable items, 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.gosu.interval

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

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.gosu.interval

uses gw.lang.reflect.interval.IterableInterval

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.gosu.interval.Color
uses example.gosu.interval.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 code prints:

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

See also

- “Custom iterable intervals using sequenceable items” on page 133
- “Enumerations” on page 169

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.

chapter 7

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. Instead of writing Java code for your Gosu to call, if you write that code directly in Gosu, you can do everything that you can do in Java. For example, you can 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 and how to write Java code that works with Gosu and works with Guidewire products.

See also

- "Gosu introduction" on page 19
- "Notable differences between Gosu and Java" on page 46
- *Integration Guide*

Overview of writing Gosu code that calls Java

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

```
var bd = new java.math.BigDecimal(5, 25)
```

Gosu provides additional optional features and more concise syntax than Java, such as optional type inference and optional semicolons.

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 converts between primitives and the equivalent object types automatically in most cases.
- Call methods on Java types.

For methods that look like getters and setters, Gosu exposes the methods also as Gosu properties.

- Get instance variables and static variables from Java types.
- Add new methods to Java types by using Gosu enhancements.
- Add new properties to Java types by using Gosu enhancements.
- Add new properties to Java types automatically for methods that look like getters and setters.
- Create Gosu classes that extend Java classes.
- Create Gosu interfaces that extend Java interfaces.
- Use Java enumerations.
- Use Java annotations.
- Use Java properties files.

All 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 PolicyCenter works with entity data.

See also

- "Gosu introduction" on page 19
- "Notable differences between Gosu and Java" on page 46
- *Integration Guide*

Core Java types in Gosu

Because Gosu is built on the JVM, many core Gosu classes are Java types. For example:

- The class `java.lang.String` is the core text object class for Gosu code.
- The basic collection types in Gosu reference the Java versions. For example:

```
java.util.ArrayList
print(list.get(0))
```

Java packages in scope

All types in the package `java.lang` and `java.util` are always 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)
```

The following simpler syntax produces more readable code:

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

See also

- "Importing types and package namespaces" on page 114

Using Java types in Gosu

Gosu can access all members of a Java type: variables, functions, and properties. You access these members on a Java class or object in the same way as you access members on a Gosu type, by using dot notation syntax. For Java methods that look like getters and setters, Gosu exposes the methods also as Gosu properties.

Java get, set, and 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 the 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 it 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 method 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 it as a getter for the property. The property name matches the original method but without the prefix `get`. For example, Gosu exposes the Java method with signature `getName()` as a `property get` function for the property named `Name` of type `String`.
- Similar to the rules for `get`, if the method name starts with `is`, takes no arguments, and returns a Boolean value, Gosu exposes it as a property accessor, a getter. The property name matches the original method but without the prefix `is`. For example, Gosu exposes the Java method signature `isVisible()` as a `property get` function for the property named `Visible`.
- Gosu does these transformations on static methods as well as on instance methods.
- Even though Gosu provides a new property that you can access for reading and writing information, Gosu does not remove the getter and setter methods. For example, if a Java object has an `obj.getName()` method, you can use the expression `obj.Name` or `obj.getName()`.

If the class has 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 remain 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 arg 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 assignments cause syntax errors if you uncomment them.
// They are not writable—there is no setter method.
//
// c.Area = 3
// c.Circumference = 4
// c.Round = false

// These Java methods do not convert to properties because
// they have the wrong number of arguments or the wrong types.
c.isMethod1()
var temp2 = c.getMethod2(1,2)
c.setMethod3()
var temp4 = c.setMethod4(8,9)
```

This Gosu code prints 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()
```

Static members and static import in Gosu

Gosu supports static members on a type: variables, functions, and property declarations. A static member exists only on the type itself, not on instances of the type. You access static members on Java types just as you would on native Gosu types.

For Gosu code that accesses static members, you qualify the class that declares the static member or import the static members from the class. For example, the following line uses the cosine function and the static reference to value `PI` from the `Math` class without importing the static members:

```
var cosHalfPiDgrees = Math.cos(Math.PI * 0.5)
```

To use the static members without qualifying them with the class name, you use the syntax of the `uses` statement that imports static members. For example, the following lines are equivalent to the previous, single line of code:

```
uses java.lang.Math#*
var cosHalfPiDgrees = cos(PI * 0.5)
```

See also

- "Importing static members" on page 116

Using Java generics

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

The runtime information about a type that uses generics is different for Java and Gosu generic types. You can access this information by using the `typeof` operator:

- The Java compiler erases the parameterization of the type during the compilation to Java byte code. At run time, the type `ArrayList<Integer>` for a Java type appears just as `ArrayList`. This Java behavior is known as type erasure.
- If the type is defined in Gosu, the runtime type retains the parameterization of the generic type. At run time, the type `ArrayList<Integer>` for a Gosu type appears as `ArrayList<Integer>`. This Gosu behavior is known as type reification.

Using other Java features in Gosu

You can use many other Java features in Gosu.

Implementing Java 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.

See also

- "Interfaces" on page 171

Using Java enumerations

Gosu can directly use Java enumerations.

Gosu also supports its own native enumeration definitions.

See also

- "Enumerations" on page 169

Using Java annotations

Gosu can directly use Java annotations.

Gosu also supports its own native annotation definitions.

See also

- "Annotations" on page 209

Using Java primitives

Gosu supports the following primitive types: `int`, `char`, `byte`, `short`, `long`, `float`, `double`, `boolean`, and the special value `null` that means an empty object value. These primitives are the full set that Java supports. The Gosu versions of these primitives 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. Java does the same for its primitive types. For example, for `int` there is the `java.lang.Integer` type that descends from the `Object` class.

- The category of object types that represent the equivalent of primitive types are called *boxed primitive* types.
- Primitive types are also called *unboxed primitives*.

The boxed primitive Java classes can be freely used in Gosu code because of Gosu's special relationship to the Java language.

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 the differences are important.

In both Gosu and Java, the language primitive types like `int` and `boolean` work differently from objects, which are descendents of the root `Object` class. For example:

- You can add objects to a collection, but you cannot add primitives.
- Variables typed to an object type can have the value `null`, but variables typed to primitives cannot.

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 unboxed types, a feature that is called *unboxing*.

In most cases, Gosu handles differences between boxed and unboxed types for you by automatically converting values as required. For example, Gosu implicitly converts between the native language primitive type called `boolean` and the Java-based object class `Boolean` (`java.lang.Boolean`). If you want explicit coercion, 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 and 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
```

Using a Java properties file

Gosu provides a utilities class, `gw.util.PropertiesFileAccess`, that supports parsing a Java properties file. By using this class, you can access the values of the properties in a straightforward way.

You use the `getProperties` method of the `PropertiesFileAccess` class to load the properties from the file as a Java `Properties` object. This method takes the full path and name of the properties file as a `String` parameter. For example, if your properties file is `GlobalProperties.properties` in `com.mycompany.properties`, you pass `com/mycompany/properties/GlobalProperties.properties` to `getProperties`.

You then use the Java `getProperty` method on the `Properties` object to get each Java `Property` object. You can convert the Java `Property` object into a static Gosu property.

Example: Accessing Java properties in Gosu

Consider the following Java properties file, `MyProps.properties`, which is located in the `gsrc.doc.examples` folder in Guidewire Studio.

```
# doc.examples.MyProps

# The hash character as the first char means that the line is a comment
! The exclamation mark character as the first char means that the line is a comment

website =  http://guidewire.com/
transport = bicycle

# 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
```

The following Gosu class converts some of the Java properties into static Gosu properties.

```
package doc.examples

uses gw.util.PropertiesFileAccess

class MyPropsAccess {
  static property get MyProperties() : Properties {
    // Using getProperties reads the file only once. The Properties are cached as a static property.
    // You must use the fully qualified path to the properties file even though MyPropsAccess and
    // MyProps.properties are in the same folder.
    return PropertiesFileAccess.getProperties("doc/examples/MyProps.properties")
  }

  static property get Website() : String {
    return MyProperties.getProperty("website")
  }

  static property get Transport() : String {
    return MyProperties.getProperty("transport")
  }

  static property get Message() : String {
    return MyProperties.getProperty("message")
  }
}
```

The following code reads the static Gosu properties.

```
// Get static properties
print("  website: ${MyPropsAccess.Website }")
print("transport: ${MyPropsAccess.Transport }")
print("  message: ${MyPropsAccess.Message }")

// Use methods on the MyProperties static property to access other property values
print("     tab: before${MyPropsAccess.MyProperties.getProperty("tab")}after")
// Access a multi-level property
print("multi-level:  ${MyPropsAccess.MyProperties.getProperty("gosu.example.properties.Key1")}")
```

Running this code produces the following output.

```
  website: http://guidewire.com/
transport: bicycle
  message: Welcome to Gosu!
      tab: before    after
multi-level: Value1
```

chapter 8

Classes

Gosu classes encapsulate data and code for a specific purpose. You can store and access data and invoke functions, known as methods, on instances of a class or on the class itself. You can subclass and extend Gosu classes to store and retrieve additional data and to invoke new methods. You can use Guidewire Studio to create and manage Gosu classes.

See also

- *Configuration Guide*

What are classes?

Gosu classes encapsulate data and code to perform a specific task. Typical use of a Gosu class is 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 invoke the Gosu functions of the class. Gosu functions defined in a class are also called *methods*.

You can *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()
```

You can also define data and methods that belong to the class itself, rather than an instance of the class. You use this technique, 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 creates an instance of the class. Gosu does not force you to choose between the two design styles.

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 also

- "Static modifier" on page 163
- "Generics" on page 221
- *Configuration Guide*

Creating and instantiating classes

In Studio, create a class by right-clicking a package name on the **Project** pane, and then clicking **New→Gosu Class**. Studio creates a simple class in which you build the functionality that you need. Studio creates the package name and class definition.

After creating a new class, add constructors, class variables, properties, and functions to the class. All of these components are optional. Within a class, functions are also called *methods*. This term is standard object-oriented terminology. This documentation refers to functions as methods if the functions are part of classes.

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 parentheses that enclose a parameter list, which is empty if the method has no arguments. The parameter list uses commas to separate multiple parameters. Each parameter has the format:

```
parameterName : typeName
```

If the method returns a value, after the parentheses, put a colon (`:`) and the return type. If the method does not return a value, specifying `void` as the return type is optional.

For example, the following line declares a method that has a single `String` parameter. The method does not return a value:

```
function doAction(arg1 : String)
```

A 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)
  }
}
```

Extending another class to make a subclass

You can extend another class to inherit methods, fields, and properties from a parent class by using the `extends` keyword after your class name, followed by the parent class name. For example:

```
class Tree extends Plant {
}
```

Your class is a subclass and subtype of the parent class. The parent class is a superclass and supertype of your class.

Constructors

A Gosu class can have a *constructor*, which is a special method that Gosu calls after creating an instance of the class. For example, the Gosu code "`new MyClass()`" calls the `MyClass` class's constructor for initialization or other actions. To create a constructor, name the method `construct`. Do not use the `function` keyword. If you do not need a class constructor, you do not need to create the `construct` method.
For example, the following class contains one constructor that takes no arguments:

```
class Tree {
  construct() {
    print("I just created a Tree object!")
  }
}
```

You can create constructors that take arguments, just as for other methods. Any code that instantiates that object can pass arguments to the `new` expression. For example, if the constructor takes one argument that is a `String` object, you can instantiate the class by using code such as the following line:

```
var x = new Tree("Name for the tree")
```

If the class constructors all take one or more arguments, any code that instantiates the class must provide a parameter list. Failing to pass arguments to a `new` expression for this class produces a compilation error.
If your class extends another class, your constructor can call the constructor of its superclass by using the `super` keyword, followed by parentheses. This statement must be the first statement in the subclass constructor. For example, the following code calls the superclass constructor, and then performs additional work.

```
class Tree extends Plant {
  construct() {
    super()
    print("I just created a 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 that you write, Gosu always calls a version of the superclass constructor, even if you omit an explicit reference to `super()`. If your class does not call the superclass constructor explicitly, Gosu attempts to use the no-argument superclass constructor:

- If a no-argument constructor exists in the supertype, Gosu calls that constructor, which is equivalent to the code `super()`.
- If a no-argument constructor does not exist in the supertype, Gosu throws a `no default constructor` compilation syntax error.

Static methods and variables

If you want to call a method directly on the class itself rather than an instance of the class, you can provide a static method. To declare a method as static, add the keyword `static` before the function declaration. You use similar syntax to make a variable static. For example, this Gosu code declares an instance method:

```
public function doAction(arg1 : String)
```

This Gosu code declares a static method:

```
static public function doAction(arg1 : String)
```

Gosu does not support a class having both an instance method and a static method that have the same name and argument list.

See also

- "Static modifier" on page 163

Using public properties instead of public variables

Although Gosu supports public variables for compatibility with other languages, best practice is to use public properties backed by private variables instead of using public variables. To define the public property name to use to access the data, you use the syntax `as` *`PROPERTY_NAME`* at the end of the declaration.

For example, to declare a public property, use this style of variable declaration:

```
private var _firstName : String as FirstName
```

Do not declare a public field:

```
public var FirstName : String   // Do not do this. Public variables are not standard Gosu style
```

See also

- "Properties" on page 151

Creating a new instance of a class

Typically, you define a class, and then create one or more instances of that class. An instance of a class is also known as an *object*. Each instance has its own set of associated data. The process of constructing a new 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 use object initializers to set properties on an object immediately after a `new` expression. Use object initializers for compact and clear object declarations. Object initializers are especially useful in combination with data structure syntax and nested objects. A simple use of an object initializer looks like the following line:

```
var sampleClaim = new Claim() {:ClaimId = "TestID"}  // Initialize the ClaimID on the new claim.
```

Note: You can use static methods, static variables, and static properties of a Gosu class without creating an instance of the class.

See also

- "Using the operator new in object expressions" on page 86
- "Static modifier" on page 163

Naming conventions for packages and types

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.

The following restrictions apply to package names.

- Package names use only lowercase characters.
- The root package name must not be a Gosu or Java strong keyword, such as `int`.
- Any subsequent part of the package name can be any name that conforms to Gosu naming conventions. You can use any Gosu or Java strong keyword. If you use a Java strong keyword, Studio displays a warning message that

you will not be able to create classes in the package. This warning applies only to Java classes. You can create a Gosu class in the package.

Class names or other type names must always start with a capital letter. Type names can contain additional capital letters later in the name for clarity. If you write Gosu enhancements, the naming rules are slightly different from the conventions for other types.

Use the following standard package naming conventions:

Type of class	Package	Example of fully qualified class name
Classes you define	*customername.subpackage*	`smithco.messaging.QueueUtils`
Guidewire Professional Services classes	`gwservices.`*subpackage*	`gwservices.messaging.QueueUtils`

WARNING Your types must never use a package name with the prefix "`com.guidewire.`" or the prefix "`gw.`" Those package namespaces are reserved for internal types.

See also

- "Importing types and package namespaces" on page 114
- "Enhancement naming and package conventions" on page 204

Using uses lines to import a type

You can add `uses` lines to your Gosu type to import types so that you can refer to them later without specifying the fully qualified name. For example, by including the following line, you can refer to the `FileInputStream` class as `FileInputStream`, rather than `java.io.FileInputStream`:

```
uses java.io.FileInputStream
```

The `uses` lines must be at the top of the class, not within a Gosu method definition.

See also

- "Importing types and package namespaces" on page 114
- "Importing static members" on page 116

Properties

Gosu classes can define properties for other objects to access the property value similarly to variables on the class. Other objects use the intuitive syntax of the period symbol (`.`) to access the properties. You can use Gosu code to implement get and set functionality for a property. The simplest code that gets or sets a property value uses an instance variable, but you can implement properties in other, more dynamic ways.

To get and set properties on an object that has `Field1` and `Field2` properties, use the period symbol just as for getting and setting standard variables:

```
// Create a new class instance
var a = new MyClass()

// Set a property value
a.Field1 = 5

// Get a property value
print(a.Field2)
```

The most straightforward form of a property definition is like a function that uses the keywords "`property get`" or "`property set`" before the function name instead of "`function`". The `get` property function must take zero parameters and the `set` property function must take exactly one parameter.

For example, the following code defines a `Field3` property that supports both set and get functionality:

```
class MyClass {
  property get Field3() : String {
    return "myFirstClass" // In this simple example, do not 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 typical class, you use a private instance variable to store the value:

```
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 needs to use the data uses the period symbol (`.`) and 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
```

Code within a class does not need to use the property name to access a property on the same class. Classes can access their own private variables. In the previous example, other methods in that class could reference the `_field4` variable rather than using the property accessor `Field4`.

Your property getter and setter methods can perform complex calculations or store the data in some other way than as a variable.

Variable alias syntax

Gosu variable alias syntax provides a shortcut to declare an instance variable as a property. *Variable alias* syntax uses the `as` keyword followed by the property name 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
}
```

Standard Gosu style is to use public properties backed by private variables instead of using public variables.

Write your Gosu classes to declare properties similar to the following line:

```
private var _firstName : String as FirstName
```

This code declares a private variable called `_firstname`, which Gosu exposes as a public property called `FirstName`.

Do not write your classes to declare public variables like the following line:

```
public var FirstName : String
```

Read-only properties

By default, properties are both readable and writable, but you can make a property read-only by adding the keyword `readonly` before the property name, as shown in the following code:

```
class MyClass {
  private var _firstname : String as readonly FirstName
}
```

Properties are dynamic and virtual functions that act like data

In contrast to standard instance variables, `property get` and `property set` functions are virtual. You can override *virtual* property functions in subclasses and implement them from interfaces. The following code shows how you override a property in a subclass. You can even call the superclass's get or set property function:

```
class MyClass {
  var _propertyVar : String as MyProperty
}

class MySubClass extends MyClass {
  override property get MyProperty() : String {
    return super.MyProperty + " from MySubClass"
  }
}
```

The overridden `property get` function first calls the implicitly defined get function from the superclass, which gets the class variable called `_propertyVar`, and then appends a string. This get function does not change the value of the class variable `_propertyVar`, but code that accesses the `MyProperty` property from the subclass gets a different value.

For example, create the classes in the previous code. Next, write and then run the following code in the Gosu Scratchpad:

```
var f = new MyClass()
var b = new MySubClass()

f.MyProperty = "MyPropValue"
b.MyProperty = "MyPropValue"

print(f.MyProperty)
print(b.MyProperty)
```

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. Only expressions separated by period (`.`) characters that access a series of instance variables or properties support null safety, such as the following 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 tend to simplify real-world code. Often, a `null` expression result has the same meaning whether the final property access is `null` or earlier parts of the path are `null`. For such cases in Gosu, do not check for the `null` value at every level of the path. This conciseness makes your Gosu code easier to read and understand.

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

```
house.Walls.Windows
```

In some languages, you must be aware that if `house` is `null` or `house.Walls` is `null`, your code throws a null-pointer exception. The following common coding pattern avoids the exception:

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

// Check earlier parts of the path for null to avoid a null-pointer exceptions (NPE)
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`. The following Gosu code is equivalent to the previous example and a null-pointer exception never occurs:

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

By default, method calls are not null-safe. If the right side of a period character is a method call, Gosu throws a null-pointer exception (NPE) if the value on the left side of the period is `null`.
For example:

```
house.myaction()
```

If `house` is `null`, Gosu throws an NPE. Gosu assumes that method calls might have side effects, so Gosu cannot skip the method call and return `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 the period: `?.`
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

A *null-safe method call* does not throw an exception if the left side of the period character evaluates to `null`. Gosu just returns `null` from that expression. Using the `?.` operator calls the method with `null` safety:

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

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

Null-safe versions of other operators

Gosu provides null-safe versions of other common operators:

- The null-safe default operator (`?:`). You use this operator to specify an alternative value if the value to the left of the operator evaluates to `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. The expression 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 by 2, or return null if cost is null
  ```

Design code for null safety

Use null-safe operators where appropriate. These operators make code more concise and simplify the handling of edge cases.

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

IMPORTANT Expose public data as properties rather than as getter functions. By using a property, you to take advantage of Gosu null-safe property accessor paths. Standard Gosu practice is to separate your implementation from your class's interaction with other code by using properties rather than public instance variables. Gosu provides the `as` keyword to expose an instance variable as a property.

See also

- "Handling null values in expressions" on page 96

Designing APIs around null-safe property paths

You may want to design your Gosu code logic around the `null`-safety feature. For example, Gosu uses the `java.lang.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 method as the `Empty` property. Use of this property is difficult with Gosu property accessor paths. For example, consider the following `if` statement:

```
if (obj.StringProperty.Empty)
```

Because `null` coerces implicitly to the type `Boolean`, which is 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 value of `StringProperty` is non-null but its `Empty` property evaluates to `false`.

In typical code, distinguishing between these two very different cases is important. For example, if you need to use the value `obj.StringProperty` only if the value is non-empty, just checking the value `obj.StringProperty.Empty` is insufficient.

To work around this restriction, Gosu adds an enhancement property to `java.lang.String` called `HasContent`. This property effectively has the reverse of the logic of the `Empty` property. The `HasContent` property only returns `true` if the property has content, so you can use property accessor paths such as the following line:

```
if (obj.StringProperty.HasContent)
```

Because `null` coerces implicitly to the type `Boolean`, which is 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 and so its `HasContent` property evaluates to `false`.

These cases are much more similar semantically than the variant that uses `Empty` (`obj.StringProperty.Empty`).

Be sure to consider null-safety of property paths as you design your code, particularly with Boolean properties.

Static properties

You can use static properties directly on the class without creating an instance of the class.

See also

- "Static modifier" on page 163

More property examples

The following examples demonstrate how to create and use Gosu class properties and their getter and setter methods.

The examples use 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"
  }
}
```

Create the classes, and then try the following lines in Gosu Scratchpad to test these classes.

First, create an instance of your class:

```
var test = new mypackage.MySecondClass()
```

Assign a property value. This code internally calls a hidden method to assign `"hello"` to variable `_f0`:

```
test.F0 = "hello"
```

The following line is invalid because `f1` is read-only:

```
// This line gives a compile error.
test.F1 = "hello"
```

Get a property value. This code indirectly calls the `mySecondClass` property getter function for `F2`:

```
print( test.F2 ) // Prints null because the property is not set yet
```

The following line is invalid because `F2` is not visible outside of the package namespace of `MySecondClass`. The `F2` property is publicly read-only.

```
// This line gives a compile error.
test.F2 = "hello"
```

Print the `Calculation` property:

```
print( test.Calculation ) // Prints 88
```

The following line is invalid because `Calculation` is read-only. This property does not have a setter function:

```
// This line 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"
```

Modifiers

Gosu supports several types of modifiers.

Access modifiers

You can use access modifier keywords to set the level of access to a Gosu class, interface, enumeration, or a type member, which is 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 access modifiers apply only to type members (functions, variables, properties, and inner types). Other access modifiers apply to both type members and top-level types (outer Gosu classes, interfaces, enumerations). The following table lists the Gosu access modifiers and their applicability and visibility:

Modifier	Description	Applies to top-level types	Applies to type members	Visible in class	Visible in package	Visible in subclass	Visible to all
`public`	Fully accessible with no restrictions	Yes	Yes	Yes	Yes	Yes	Yes
`protected`	Accessible only to types in the same package and subtypes	--	Yes	Yes	Yes	Yes	--
`internal`	Accessible only in same package	Yes	Yes	Yes	Yes	--	--
`private`	Accessible only in the declaring type, such as the Gosu class or interface that defines it	--	Yes	Yes	--	--	--

If you do not specify a modifier, Gosu uses the following default access levels:

Element	Default modifier
Types and classes	`public`
Variables	`private`
Functions	`public`
Properties	`public`

Coding style recommendations for variables

Always prefix `private` and `protected` class variables with an underscore character (_).

Avoid public variables. If you want to use public variables, convert the public variables to properties. This conversion separates the way code in other classes accesses the properties from the implementation of the storage and retrieval of the properties.

See also

- “Coding style” on page 465

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 insert the modifier if the `override` keyword 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, which contain 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 code is a simple abstract class:

```
abstract class Vehicle {
}
```

If a type is specified as abstract, Gosu code cannot construct an instance of that type. For example, you cannot use code such as `new MyType()` for an abstract type. You can instantiate a subtype of the abstract 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 called 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 the method, that method would be declared `abstract`. Another class `B` could extend `A` and provide code to implement that abstract method. Class `A` is the abstract class and class `B` is a concrete subclass of `A`.

For example, the following code defines an abstract Gosu class called `Vehicle`, which contains members but no abstract members:

```
package com.mycompany

abstract class Vehicle {
  var _name : String as Name
}
```

You cannot construct an instance of this class, but you can define another class that extends the class:

```
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 a concrete implementation of each abstract method or property to support instantiation 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 name.

The subtype must implement an abstract function or abstract property using the same name as the supertype. Use the `override` keyword to tell Gosu that the subtype overrides an inherited function or method of the same name. If you omit the `override` keyword, Gosu displays a compiler warning. Additionally, the Gosu editor offers to insert the `override` modifier if the `override` keyword seems appropriate.

For example, the following code expands 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` class looks like the following code:

```
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.
   * This variable provides a concrete implementation of the abstract property "Plate".
   */
  var _licenseplate : String as Plate

  /* Implement the function RegisterWithDMV, which is abstract in the supertype, which
   * does not define how to implement the method, but does specify the method signature
   * that you must implement to support instantiation with "new".
   */
  override function RegisterWithDMV(registrationURL : String ) {
    // Here do whatever needs to be done
    print("Simulation of registering " + _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 that type is abstract.

Create the classes, and then try the following lines in Gosu Scratchpad to test these classes.

```
var t = new com.mycompany.Truck()
t.Plate = "ABCDEFG"
print("License plate = " + t.Plate)
t.RegisterWithDMV( "http://dmv.ca.gov/register" )
```

This code prints the following lines:

```
License plate = ABCDEFG
Simulation of registering 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 a subclass of that class.

The `final` modifier is implicit with enumerations. An *enumeration* is an encapsulated list of enumerated constants. Enumerations are implemented like Gosu classes in most ways. No Gosu code can subclass an enumeration.

See also

- “Enumerations” on page 169

Final class variables

Using the `final` keyword on class variables specifies that the variable can be set only once, and only in the declared class. Subclasses cannot set the value of a final class variable.

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 that variable in all possible code paths.
For example, the following syntax is valid because all constructors initialize the final variable once in each code path:

```
class TestABC {
  final var _name : String

  construct() {
    _name = "hello"
  }
  construct(b : boolean){
    _name = "there"
  }
}
```

The following code is invalid because one constructor does not initialize the final variable:

```
class TestABC {
  final var _name : String // INVALID CODE, ALL CONSTRUCTORS MUST INITIALIZE _name 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 on 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 that the superclass declares as `final`.
For example, the following code defines a class with a final function and final properties:

```
package com.mycompany

class Auto {

  // A final property -- no subtype can reimplement or override this property!
  final property get Plate() : String
  final property set Plate(newPlate : String)

  // A final function/method -- no concrete subtype can reimplement or override this function!
  final function RegisterWithDMV(registrationURL : String)
}
```

See also

- “Properties” on page 151

Final local variables

You can use the `final` modifier on a local variable to initialize the value and prevent the value from changing.
For example, the following code is valid:

```
class final1 {
  function PrintGreeting() {
```

```
    var f = "frozen"
    f = "dynamic"
  }
}
```

This code is not valid because it attempts to change the value of the final variable:

```
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 the value of the variable immediately in the declaration. Alternatively, you can declare the variable as `final` but not immediately initialize the variable with a value. You must set the value in that function for all possible code paths. For example, any `if` statements or other flow control structures must set the value of the variable and not change a value that previous code set. The Gosu compiler verifies that all code paths initialize the final variable exactly once.

For example, the following code is valid:

```
function finalVar(a : boolean) {
  final var b : int
  if (a) {
    b = 0
  } else {
    b = 1
  }
}
```

If you remove the `else` branch, the code is not valid because the final variable is initialized only if `a` is `true`.

```
function finalVar(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 on a function parameter to prevent the value of the parameter 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 this method with the code:

```
var f = new example.FinalTest()
var s = "Butter"
f.SuffixTest( s )
```

This code prints:

```
Butterfly
```

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 FinalTest {
  function SuffixTest(final greeting : String) {
    greeting = greeting + "fly"
    print(greeting)
  }
}
```

Static modifier

Gosu supports static variables, functions, properties, and inner types.

Static variables

Gosu classes can define a variable as static. A *static variable* is stored once for a Gosu class, rather than once for each instance of the class. The `static` modifier can be used with variables and properties.

WARNING If you use static variables in a multi-threaded environment, you must take 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 additional help on static variables and synchronized access, contact Customer Support.

To use a Gosu class variable, set its access level, such as `internal` or `public`, to restrict access to the set of classes that need to use the variable.

The `static` modifier cannot be combined with the `abstract` modifier.

Static functions and properties

The `static` modifier can also be used with functions and properties to indicate that the function or property 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.

The following code accesses properties on the class itself, not an instance of the class. You can create the class, and then test this code in the Gosu Scratchpad:

```
Greeting.Name = "initial value"
print(Greeting.Name)
Greeting.PrintGreeting()
```

This example does not use the `new` keyword to construct an instance of the `Greeting` class.

Static inner types

The `static` modifier can also be used with inner types, such as inner classes. The `static` modifier on the inner type indicates that the inner type belongs to the outer type itself rather than instances of the type.

An inner class must be declared `static` in order to define static members, such as static methods, static fields, or static properties. This Gosu requirement mirrors the requirement of Java inner classes.

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()
  }
}
```

Create the classes, and then try the following lines in Gosu Scratchpad to test the inner class.

```
print(example.Greeting.Hello)
```

This code prints:

```
Bonjour
```

See also

- “Access modifiers” on page 157
- “Abstract modifier” on page 159
- “Inner classes” on page 164
- “Concurrency” on page 417

Inner classes

You can define inner classes in Gosu, similar to inner classes in Java. Inner classes are useful for encapsulating code even further within the same class as related code. Use named inner classes if you need to refer to the inner class from multiple related methods or multiple related classes. Use anonymous inner classes if you need a simple subclass that you can define in-line within a class method.

Inner classes can optionally include generics features.

The outer class that defines the inner class is also called the *enclosing class*.

See also

- “Generics” on page 221

Named inner classes

You can define a named class within another Gosu class. This inner class can be used within the outer class that contains the definition, or from classes that extend the outer class.

Example

The following code demonstrates a simple inner class.

```
package example

 class Greeting {
  // function in the main (outer) class
  public function greet() {
    var x = new FrenchGreeting()
    print(x.sayWhat())
  }

   // INNER CLASS DECLARATION
  public class FrenchGreeting {
    public function sayWhat() : String {
      return "Bonjour"
    }
  }
}
```

Create the classes, and then try the following lines in Gosu Scratchpad to test the inner class.

```
var f = new example.Greeting()
f.greet()
```

This code prints:

```
Bonjour
```

Referring to members of the outer class

Within the inner class, you refer to methods, fields, and properties on the outer class by using the `outer` keyword followed by a period, followed by the property name.

The following example has an inner class that refers to a field on the outer class:

```
package example

class Greeting {
  // Field in the main (outer) class
  public var intro : String = "I would like to say"

  // Function in the main (outer) class
  public function greet() {
    var x = new FrenchGreeting()
    print(x.sayWhat())
  }

  // INNER CLASS DECLARATION
  public class FrenchGreeting {
    public function sayWhat() : String {
      return outer.intro + " " + "bonjour"
    }
  }
}
```

If there is no ambiguity about the variable to which a name refers, the `outer` keyword and the following period are optional:

```
return intro + " " + "bonjour"
```

Static inner classes

A *static inner class* is an inner class that is a singleton, which is a predefined single instance within its parent class. You do not instantiate this class.

The following example defines a static inner class called `FrenchGreeting` within the `Greeting` class:

```
package example
```

```
class Greeting {
  public property get Hello() : String {
    return FrenchGreeting.sayWhat()
  }

  // STATIC INNER CLASS DECLARATION
  static class FrenchGreeting {
    static public function sayWhat() : String {
      return "Bonjour"
    }
  }
}
```

You can test this class in the Gosu Scratchpad using the code:

```
print(example.Greeting.Hello)
```

This code prints:

```
Bonjour
```

Notice that this example does not construct 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.

An inner class must be declared `static` in order to define static members, such as static methods, static fields, or static properties. This Gosu requirement mirrors the requirement of Java inner classes.

See also

- “Static modifier” on page 163

Classes that extend the outer class can use the inner class

Classes that derive from the outer class can use the inner class.

The following code defines the `Greeting` class and its inner class `FrenchGreeting`.

```
package example

class Greeting {
  // function in the main (outer) class
  public function greet() {
    var x = new FrenchGreeting()
    print(x.sayWhat())
  }

  // INNER CLASS DECLARATION
  public class FrenchGreeting {
    public function sayWhat() : String {
      return "Bonjour"
    }
  }
}
```

The following example subclasses the `Greeting` class and uses the method in the inner `FrenchGreeting` class:

```
package example

class OtherGreeting extends Greeting {
  public function greetme() {
    var f = new Greeting.FrenchGreeting()
    print(f.sayWhat())
  }
}
```

You can test this code by using the following code in the Gosu Scratchpad:

```
var t = new example.OtherGreeting()
t.greetme()
```

This code 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 creating a named inner class. Constructing instances of anonymous inner classes is similar in many ways to using the `new` operator to create instances of a base class. However, you can extend the base class by following the class name with braces (`{}`) and then adding more variables or methods. If you do not have a more useful base class, use `Object`.

Example 1

The following class 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 Scratchpad to test this class:

```
example.InnerTest.runme()
```

This code prints:

```
Value is 0 at creation!
Value is 1
Value is 2
Value is 3
Value is 4
Value is 5
```

Example 2

The following example demonstrates an advanced anonymous inner class. This anonymous inner class derives from a more functional class than `Object`. In this example, new inner class inherits the constructor and another method.

The following code defines a base class named `Vehicle` for the inner class:

```
package example

class Vehicle {
  construct() {
    print("A vehicle was just constructed!")
  }

  function actionOne(s : String) {
    print("actionOne was called with arg " + s)
  }
}
```

The following code creates a different class that uses `Vehicle` and defines an anonymous inner class that is 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" )
  }
}
```

The inner class that defines the `actionTwo` method uses the `new` operator and not the `class` operator. The `new` operator defines a new class with no name and then creates one instance of that class.

Create the classes, and then try the following lines in Gosu Scratchpad to test the `FancyVehicle` class:

```
var g = new example.FancyVehicle()
g.testInner()
```

This code 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 shortcut is especially helpful for APIs defined to take the type `BlockRunnable` or `Runnable`.

See also

- “Blocks as shortcuts for anonymous classes” on page 181

chapter 9

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.

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 the manufacturers as named constants that can be checked at compile time. Gosu supports enumerations natively and also is compatible with enumerations defined in Java.

Create an enumeration

An enumeration provides a fixed set of values that you can use for a property or variable. For example, a `FruitType` enumeration provides names of fruits.

Procedure

1. Create an enumeration by using the same approach that you use to create a class.
   a. In Studio, right-click a package folder and click the **New** submenu.
   b. Click **Class** to create the enumeration class in that package.
   c. Type a name for the enumeration.
   d. In the drop-down list, click **Enum**.

   Your enumeration looks like:

```
package example

enum FruitType {

}
```

2. Add your named constants separated by commas:

```
enum FruitType {
  Apple, Orange, Banana, Kiwi, Passionfruit
}
```

Extracting information from an enumeration

To use an enumeration, use the period (.) operator to reference elements of the enumeration class:

```
enumerationClassName.enumerationConstantName
```

To use the name of the enumeration element as a `String`, get its `Name` property. To use the zero-based index of the enumeration element as an `Integer`, get its `Ordinal` property.

Extract information from an enumeration

You can assign the name of a value from an enumeration to a variable. You use dot notation to access the name of the enumeration value.

Before you begin

To create the enumeration that this example uses, you must perform the steps in the following topic:

- "Create an enumeration" on page 169

Procedure

1. In Studio, in Scratchpad, create a variable that contains an enumeration constant:

```
uses example.FruitType
var myFruitType = FruitType.Banana
```

2. Add the following lines to access the properties of the enumeration constant:

```
print(myFruitType.Name) // Prints "Banana"
print(myFruitType.Code) // Prints "Banana"
print(myFruitType.Ordinal) // Prints "2"
```

3. Run the program. The following lines appear in the console:

```
Banana
Banana
2
```

Comparing enumerations

You can compare two enumerations by 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.")
}
```

Running this code produces the following line if `myFruitType` references `FruitType.Banana`:

```
Watch out for banana peels.
```

See also

- "Create an enumeration" on page 169
- "Extract information from an enumeration" on page 170

chapter 10

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.

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, but leave 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.

A physical example of an interface is 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. Because 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, volume up, and volume down.

Similarly, PolicyCenter defines interfaces that PolicyCenter calls to perform various tasks or calculate values. For example, to integrate PolicyCenter 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.

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 provides 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 either Gosu or Java.

See also

- *Integration Guide*

Defining and using an interface

In some ways, interfaces are similar to Gosu classes. There is no separate top-level folder in Studio for interfaces. You use the `implements` keyword in a class declaration to specify the interfaces that the class implements. If a class implements more than one interface, separate the interface names by commas:

```
class MyRestaurant implements Restaurant, InitializablePlugin
```

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 interface methods that look like properties must be implemented in Gosu explicitly as a Gosu property. In other words, if the interface contains a method whose name starts with `"get"` or `"is"` and takes no parameters, define the method using the Gosu `property` syntax. In this case, do not use the `function` keyword to define the method as a standard class method.

For example, if interface `IMyInterface` declares methods `isVisible()` and `getName()`, your implementation of this interface might look like:

```
class MyClass implements IMyInterface {

  property get Visible() : Boolean {
  ...
  }
  property get Name() : String {
  ...
  }
}
```

Gosu interfaces can extend from Java interfaces. You can also have your interface include Gosu generics. Your interface can extend from a Java interface that supports generics. Your interface can abstract an interface across a type defined in Java or a subtype of such a type.

See also

- “Generics” on page 221

Create and implement an interface

Procedure

1. To create an interface in Guidewire Studio, first create an appropriate package.
   a. Right-click the package folder, and then click the **New** submenu.
   b. Click **Class** to create the class in that package.
   c. Type a name for the interface.
   d. In the drop-down list, click **Interface**.
      Studio creates a new interface with the appropriate syntax.
2. Write the rest of the interface like a Gosu class, except that methods are method signatures only with no method bodies.
   For example, define an interface with the following code:

   ```
   interface Restaurant {
     function retrieveMeals() : String[]
     function retrieveMealDetails(dishname : String) : String
   }
   ```

3. To implement an interface, create a Gosu class and add "`implements` *`MyInterfaceName`*" after the class name.
   For example, if your class is called `MyRestaurant`, go to the line:

   ```
   class MyRestaurant
   ```

4. Change that line to:

```
class MyRestaurant implements Restaurant
```

5. For the example `Restaurant` interface, implement the interface with a class.
   For example, use code like the following lines:

```
class MyRestaurant implements Restaurant

  override function retrieveMeals() : String[]{
    return {"chicken", "beef", "fish"}
}
  override function retrieveMealDetails(dishname : String) : String {
    return "Steaming hot " + dishname + " on rice, with a side of asparagus."
  }
}
```

Defining and using properties on interfaces

Interfaces created in Gosu can declare properties by specifying 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, the following code defines an interface:

```
package example

interface MyInterface {
  property get VolumeLevel() : int
  property set VolumeLevel(vol : int) : void
}
```

A class could implement this interface with the following code:

```
class MyStereo implements MyInterface {
  var _volume : int

  property set VolumeLevel(vol : int) {
    _volume = vol
  }

  property get VolumeLevel() : int {
    return _volume
  }
}
```

After creating the class and interface, you can test the following code in the Gosu Scratchpad:

```
uses example.MyStereo

var v = new MyStereo()
v.VolumeLevel = 11
print("The volume goes to " + v.VolumeLevel)
```

This code prints:

```
The volume goes to 11
```

Alternatively, a class implementing a property can implement the property with the *variable alias* syntax by using the `as` keyword. Using this language feature, you make set methods that use a class instance variable to store property values, and get methods to get property values from the variable.

For example, the following, more concise code is functionally identical to the previous example implementation of `MyStereo`:

```
uses example.MyStereo

class MyStereo implements MyInterface {
  var _volume : int as VolumeLevel
}
```

If you run the same Gosu Scratchpad code as previously, the code prints the same results.

Interface methods that look like properties

If an interface's methods look like properties, a class that implements the interface must implement those methods in Gosu as a property by using `property get` or `property set` syntax. If the interface contains a method whose name starts with `"get"` or `"is"` and takes no parameters, define the method by using the Gosu `property` syntax. See earlier in this topic for examples.

See also

- "Classes" on page 147
- "Properties" on page 151

Modifiers and interfaces

In many ways, interfaces are defined like classes. One way in which they are similar is the support for modifier keywords.

One notable difference for interfaces is that the `abstract` modifier is implicit for the interface itself and all methods defined on the interface. Consequently, Gosu does not support the `final` modifier on the interface or its members.

Implementing methods

Use of the `override` modifier is optional on an implementation of an interface method. If you do use the `override` modifier, your code in a class that implements an interface is more robust against changes in method signatures in the interface.

Superclass properties

When implementing an interface and referencing a property on a superclass, use the `super.`*`PropertyName`* syntax, such as:

```
property get WholeValue() : String {
  _mySpecialPrivateVar = super.FirstHalf + super.SecondHalf
}
```

See also

- "Modifiers" on page 157

chapter 11

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 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.

What are blocks?

Gosu blocks are functions without names, which you can define in-line within another function. In other programming languages, blocks are known as *anonymous functions*. You can then pass that block of code to yet other functions to invoke as appropriate. Blocks are 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 the block either once or possibly many times with different arguments.

For example, you might want to find items within a collection that meet particular criteria, or to sort a collection of objects by certain properties. If you can describe your find or sort criteria using a small amount of Gosu code, Gosu performs the general algorithm such as sorting the collection.

Some other programming languages have similar features to blocks and call them *closures* or *lambda expressions*. If you use the Java language, notice that Gosu blocks serve some of the 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, readable 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.

Gosu code using blocks appropriately can simplify of your Gosu code. However, blocks can also cause confusion if your code uses them too frequently. If the previous list does not include your intended use, reconsider whether to use blocks. There may be a better and more conventional way to solve the problem. If you write a method that takes more than one argument that is a block, strongly consider redesigning or refactoring the 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 (`:`) similar to the definitions of arguments to a method. Next, add

a hyphen character (-) and a greater-than character (>) to form the appearance of an arrow: ->. Finally, add a Gosu expression or a statement list surrounded by braces ({}).

The syntax of a block definition is:

```
\ argumentList -> blockBody
```

The argument list (*argumentList*) is a standard function argument list, for example:

```
x : int, y : int
```

The argument list defines the parameters to pass to the block. The parameter list uses identical syntax as parameters to regular functions. In some cases, you can omit the types of the parameters, such as passing a block directly into a class method such that Gosu can infer the parameter type.

The block body (*blockBody*) can be either of the following:

- A simple expression. This expression includes anything legal on the right-hand side of an assignment statement. For example, the following line is a simple expression:

  ```
  "a concatenated string " + "is a simple expression"
  ```

- A statement list of one or more statements surrounded by braces and separated by semi-colon characters, such as the following simple single-statement statement list:

  ```
  \ x : int, y : int -> { 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 block is invalid:

  ```
  names.each( \ n -> myValue += n )
  ```

  Changing the code to the following line produces a valid block:

  ```
  names.each( \ n -> { myValue += n } )
  ```

  For multiple statements, separate the statements with a semicolon character. For example:

  ```
  \ x : int, y : int -> { var i = x + y; return i }
  ```

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

```
var square = \ x : int-> 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.

Standard Gosu style is to omit all semicolon characters in Gosu at the ends of lines. Gosu code is more readable without these optional semicolons. However, if you provide statement lists on one line, such as within block definitions, use semicolons to separate statements.

See also

- "Argument type inference shortcut" on page 178
- "General coding guidelines" on page 465

Block return types

A block definition does not need to explicitly declare a return type, which is the type of the return value of the block. The return value is statically typed even though the type is not explicitly visible in the code. Gosu infers the return type from either the expression, if you defined the block with an expression, or from the `return` statements in a statement list. This functionality makes the block appear shorter and more elegant.

For example, consider the following block:

```
var blockWithStatementBody = \ -> { return "hello blocks" }
```

Because the statement `return "hello blocks"` returns a `String` object, the Gosu compiler infers that the return type of the block is `String`.

Invoking blocks

You can invoke blocks just like normal functions by referencing a variable to which you previously assigned the block. To use a block, type the following items in the following order:

- The name of the block variable or an expression that resolves to a block
- An open parenthesis
- A series of argument expressions
- 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 variable has a type of `String` even though the code text does not show the type explicitly.

To call this block and assign the result to the variable `myresult`, 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 has two numbers as parameters and returns the result of adding those numbers:

```
var adder = \ x : int, y : int -> { 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 `int` 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 uses an even more concise syntax:

```
var adder = \ x : int, y : int -> x + y
```

Variable scope and capturing variables in blocks

A Gosu block maintains some context with respect to the enclosing statement that created the block. If code in the block refers to variables that are defined outside the block's definition but are 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 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 uses the most recent value in the original scope. This behavior continues even if the original scope exited or 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 : int -> 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, `this` can represent the current instance of a Gosu class running a method.

This capturing feature allows the block to access variables in scope at its definition. Capturing occurs even in the following cases:

- After passing the block as an argument to another function
- After the block returns to the function that defines it
- If code assigns the block to a variable and retains the variable indefinitely
- After the original scope exits or finishes

Each time the block runs, the block can access all variables declared in the original scope in which the block 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 needed. Captured variable values are not a static snapshot of the values at the time that the block was created.

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.` Code calls the block only after that change happens:

```
// Define a local variable, which is captured by a block
var x = 10

// Create the block
var captureX = \ y : int -> 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 value is NOT a snapshot of the value at the time the block was created
var z = captureX( 10 )

print(z) // Prints 30   --- Not 20!!!
```

The captured variable is shared by the original scope and the block that was created within that scope. The block references the variable itself, not merely its original value.

IMPORTANT Capturing variables in blocks is very powerful. Use this feature very carefully. Document your assumptions so that people who read your code can understand that code.

Argument type inference shortcut

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 the type for more concise code. This behavior is particularly relevant for using collection-related code that takes blocks as arguments.

For example, suppose you have this code:

```
var x = new ArrayList<String>(){"a", "b", "c"}

var y = x.map( \ str : String -> str.length )
```

You can omit the argument type (`String`). The `map` method signature allows Gosu to infer the argument type in the block from the definition of the `map` method.

You can 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 type of the list. Therefore, Gosu infers that `str` must be of type `String`. You do not need to explicitly define either the type of the arguments or the return type.

The `map` method is implemented using a built-in Gosu enhancement of the Java language `List` interface.

See also

- See “Collections” on page 183

Block type literals

Block literals are a form of type literal. Use a block literal to reference a specific *block type*. The block literal specifies the kinds of arguments that the block takes and the type of the return value.

Block types BNF notation

The formal BNF 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
```

Block types in declarations

To declare a variable to contain a block, the preferred syntax is:

```
var variableName( list_of_argument_types ) : return_type
```

For example, to declare that `strBlock` is a variable that can contain a block that takes a single `String` argument and returns a `String` value, use this code:

```
var strBlock( String ) : String
```

In declarations, you can also optionally use the `block` keyword, although this is discouraged in declarations:

```
var variableName : block( list_of_types ) : return_type
```

For example, this code declares the same block type as described earlier:

```
var x : block( String ) : String
```

Usage of block types not in declarations

For a block type literal that is not part of a declaration, the `block` keyword is strictly required:

```
block( list_of_types ) : return_type
```

For example:

```
return b as block( String ) : int
```

The `b` variable is assigned a value that is a block type. Because the block type literal is not directly part of the declaration, Gosu requires the `block` keyword.

Block types in argument lists

A function definition can specify a function argument as a block. As you define the block argument, provide a name for that block parameter so that you can use the block within the function. Use 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 takes one argument, which is a block. Suppose the block takes a single `String` argument and returns no value. To refer to this block by name as `myCallback`, define the argument using the syntax:

```
myCallBack( String ) : void
```

Example of a block type in an argument list

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` argument.

```
package mytest

class test1 {
  function myMethod( myCallBack( String ) : void ) {

    // Call your callback block and pass it a String argument
    myCallBack("Hello World")
  }
}
```

After creating the class, you can test the following code in the Gosu Scratchpad:

```
var a = new mytest.test1()
a.myMethod( \ s : String -> print("<contents>" + s + "</contents>") )
```

For 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 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>
```

Usage of block types for recursion

Gosu blocks support recursion, in which the block calls itself until reaching an end condition. To use recursion in blocks, you must name and define the block signature before you define the block code. For example, the following code defines a block that returns the factorial of an integer:

```
var bFactorial: block(int): int
bFactorial = \ x -> x > 12 or x < 1 ? -1 : (x == 1 ? 1 : x * bFactorial(x-1))
```

This code has two end conditions. The first end condition ensures that the block terminates if the result would be greater than `Integer.MAX_VALUE` or if the parameter value is less than 1. The second end condition terminates the recursion when the argument value is 1. To test this block, use code like the following:

```
print(bFactorial(6))
```

This code prints:

```
720
```

Blocks and collections

Gosu blocks are particularly useful for working with collections of objects. Blocks support concise, readable 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 `String` objects:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}
```

You can use blocks to sort the list based on various attributes, such as the length of the strings. Create a block that takes a `String` and returns the sort key, which in this case is the `String` object'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.

See also

- "Collections" on page 183

Blocks as shortcuts for anonymous classes

In some cases, you can pass a block as a method argument instead of an anonymous class instance. You can pass a block if the method argument is an implementation of a functional interface. A *functional interface* contains exactly one abstract method. For example, the Gosu interface type `BlockRunnable` or the Java interface type `Runnable` are both functional interfaces that contain a single method called `run`. Passing a block is much more concise than passing an anonymous class instance. This Gosu coding style 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 single method of the interface. The return type of the block must be the same as the return type of that method.

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`, you can probably use a block for that type.

This technique works with any interface, including interfaces defined as inner interfaces within another interface.

Example

This example demonstrates the use of a block argument for an inner functional interface that an outer interface defines. The `MyCallbackHandler` interface contains an inner functional interface called `MyCallbackHandler.CallbackBlock`. This functional interface implements a single method named `run`, similar to the `Runnable` interface. Instead of creating an anonymous class to use the inner interface, use a block that takes no arguments and has no return value.

This Java code defines the `MyCallbackHandler` interface:

```
public interface MyCallbackHandler {

  // A functional interface within this interface
  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 {

  var _callbackHandler : MyCallbackHandler
  // Create an anonymous class that implements the inner interface
  var myBlock : _callbackHandler.CallbackBlock = new MyCallbackHandler.CallbackBlock() {

    // 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)
}
```

You can code the method more concisely by using a block:

```
public function messageReceived(messageId : int) {
  _callbackHandler.execute(\ -> { /* Your Gosu statements here */ })
}
```

chapter 12

Collections

Gosu collection and list classes rely on collection classes from the Java language. Gosu collections and lists provide significant built-in enhancements over the Java classes. By using the enhanced Gosu collection and list classes, you can loop through collection items to perform actions, extract item information, or sort items with single lines of code.

Gosu provides additional initializer syntax for both lists and maps similar to Gosu's compact initializer syntax for arrays.

Basic lists

Lists in Gosu inherit from the Java interface `java.util.List` and its common Java subclasses, such as `java.util.ArrayList`.

See also

- "Sorting lists or other comparable collections" on page 190

Creating a list

To create an empty list, you use generics notation to specify the type of object that the list contains, such as in this example of an `ArrayList` of `String` objects:

```
var myemptylist = new ArrayList<String>()
```

In many cases, you need to initialize the list with data. Gosu provides a natural syntax for initializing lists similar to initializing arrays.

For example, the following line is an array instantiation:

```
var s2 = new String[ ] {"This", "is", "a", "test."}
```

In comparison, the following line instantiates a new `ArrayList`:

```
var strs = new ArrayList<String>(){"a", "ab", "abc"}
```

The previous line is shorthand for the following code:

```
var strs = new ArrayList<String>()
strs.add("a")
```

```
strs.add("ab")
strs.add("abc")
```

See also

- “Generics” on page 221

Type inference and list initialization

Because of Gosu type inference, you can use a very concise syntax to create and initialize a list.

```
var s3 = {"a", "ab", "abc"}
```

The compile-time type of the list is `java.util.ArrayList<String>`. The run-time type is different, `java.util.ArrayList<Object>`.

Gosu infers the type of the result list as the least upper bound of the components of the list. 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 behavior ensures that 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:

- Zero classes or one class, which might be `java.lang.Object`, if no other common class is found
- One or more interfaces

At compile time, the list and its elements use the compound type. You can use the list and its elements with APIs that expect those interfaces or the ancestor element.

For example, the following code creates classes that extend a parent class and implement an interface.

```
interface HasHello {
  function hello()
}

class TestParent {
  function unusedMethod() {}
}

class TestA extends TestParent implements HasHello{
  function hello() {print("Hi A!")}
}

class TestB extends TestParent implements HasHello{
  function hello() {print("Hi B!")}
}

var newList = { new TestA(), new TestB()}

print("Run-time type = " + typeof newList)
```

This code prints the following line.

```
Run-time type = java.util.ArrayList<java.lang.Object>
```

The compile-time type is a combination of:

- The class `TestParent`, which is the most specific type in common for `TestA` and `TestB`
- The interface `HasHello`, which both `TestA` and `TestB` implement

See also

- “Compound types” on page 409

Getting and setting list values

The following verbose code sets and gets `String` values from a list by 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 code in Gosu in more natural index syntax using Gosu shortcuts.

```
var strs = {"a", "ab", "abc"}
strs[0] = "b"
var firstStr = strs[0]
```

This Gosu syntax does not automatically resize lists. If a list has only three items, the following code fails at run time with an index out of bounds exception.

```
strs[3] = "b" // Index number 2 is the higest supported number
```

Basic hash maps

Maps in Gosu inherit from the Java class `java.util.HashMap`.

Creating a hash map

To create an empty map, you use generics notation to specify the type of object that the map contains. For example, define a `HashMap` that maps a `String` object to another `String` object:

```
var emptyMap = new HashMap<String, String>()
```

In many cases, you need to initialize the map with data. Gosu provides a natural syntax for initializing maps similar to initializing arrays and lists.

For example, the following code creates a new `HashMap` where `"aKey"` and `"bKey"` are keys, whose values are `"aValue"` and `"bValue"` respectively:

```
var strMap = new HashMap<String, String>(){"aKey" -> "aValue", "bKey" -> "bValue"}
```

That code is shorthand for the following code:

```
var strs = new HashMap<String, String>()
strs.put("aKey", "aValue")
strs.put("bKey", "bValue")
```

This Gosu syntax simplifies the declaration of static final data structures of this type, and produces more readable code than the equivalent Java code.

Types of the initialized map

For even more concise code, omit the `new` expression entirely, including the type:

```
var m = {"aKey" -> "aValue", "bKey" -> "bValue"}
```

Although the type name is omitted, the map is not untyped. Gosu infers that you want to create a `HashMap` by the way you use the hyphen and greater than symbol. Gosu infers the exact type of the map from the object types that you pass to the initialization.

The run-time type is always:

```
java.util.HashMap<java.lang.Object, java.lang.Object>
```

The compile-time type depends on the types of what you pass as keys and values. Gosu infers the type of the map to be the least upper bound of the components of the map. If you are consistent in the type of your keys and the types of your values, the compile-time type of the map is `HashMap<KEY_TYPE, VALUE_TYPE>`.

If you pass different types of objects to either the keys or the values, Gosu finds the most specific type that includes all of the items that you pass. Gosu finds a common ancestor class that all the objects support. If the types implement interfaces, Gosu attempts to preserve commonality of interface support in the map type. This behavior ensures that your map and its elements act as expected with APIs that rely on support for interfaces. In some cases, the resulting type at compile time is a compound type. A *compound type*, which combines zero or one classes and one or more interfaces into a single type. The purpose is for you to be able to use that list and its elements with APIs that require the interface as the argument type. At run-time, the type of the initialized map is `HashMap<Object, Object>`.

See also

- "Creating a list" on page 183
- "Generics" on page 221

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>(){"aKey" -> "aValue", "bKey" -> "bValue"}
strs.put("cKey", "cValue")
var valueForCKey = strs.get("cKey")
```

You can write this code in the more natural index syntax using Gosu shortcuts:

```
var strs = new HashMap<String, String>(){"aKey" -> "aValue", "bKey" -> "bValue"}
strs["cKey"] = "cValue"
var valueForCKey = strs["cKey"]
```

Special enhancements on maps

Just as most methods for lists in Gosu are defined as part of Java's class `java.util.ArrayList`, most of the behavior of maps in Gosu is inherited from `java.util.Map`. 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 property is 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: one of the key type and one of the value type. This method calls this block with each key/value pair in the `Map`, providing a more natural iteration over the Map.

For example:

```
var strMap = new HashMap<String, String>(){"aKey" -> "aValue", "bKey" -> "bValue"}
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 simplest constructor takes only one argument, which is a Gosu block that provides a default value. The block takes one argument, which is 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 alternative constructor if you have an existing map that you want to wrap rather than create a new map.

For example, the following code assigns values in a new map to entries that do not have values in the original map:

```
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 expansion (*.)

Gosu includes a special operator for array expansion and list expansion. The expansion operator is an asterisk followed by a period. Array expansion is valuable if you need a single one-dimensional array or list through which you can iterate.

The return value is as follows:

- If you use the expansion operator on an array, Gosu gets a property from every item in the array and returns all instances of that property in a new, single-dimensional, read-only array.
- If you use the expansion operator on a list, Gosu gets a property from every item in the list and returns all instances of that property in a new, single-dimensional, read-only list.

You can access elements in the new array or list individually with index notation, and you can iterate through the elements in a `for` loop. You cannot modify the elements of the new array or list.

The following code creates a `String` array. String objects in Gosu have a `length` property that contains the number of characters. The code uses the expansion operator on the array to create an array of string lengths for elements of the original array. Because the `length` property is of type `int`, the expansion array also is of type `int`.

```
var stringArray = new String[] {"one", "two", "three", "four"}
var lengthArray = stringArray*.length

for (element in lengthArray) {
  print(element)
}
```

The output from this code is:

```
3
3
5
4
```

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.

Suppose you wanted to get a list of the groups to which a user belongs so that you can display the display name property of each group. Suppose a `User` object contains a `MemberGroups` property that returns a read-only array of groups to which the user belongs. In this case, the Gosu syntax `user.MemberGroups` returns an array of `Group` objects, each one of which has a `DisplayName` property. To get the display name property 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 also applies to methods. Gosu uses the type on which the method runs to determine how to expand the type:

- If the original object is an array, Gosu creates an expanded array.
- If the original object is a list, Gosu creates an expanded list.

The following example calls a method on the `String` component of the `List` of `String` objects. Gosu generates a list of initials by extracting the first character in each `String`.

```
var s = {"Fred", "Garvin"}

// Generate the character list [F, G]
var charList = s*.charAt( 0 )
```

Important notes about the expansion operator:

- The generated array or list itself is always read-only from Gosu. You cannot assign values to elements of the array, such as setting `nameArray[0]`.
- The expansion operator `*.` applies only to array expansion, not for standard property accessing.
- When using the `*.` expansion operator, only component type properties are accessible.
- When using the `*.` expansion operator, array properties are not accessible.
- The expansion operator applies to arrays and to any `Iterable` type and all `Iterator` types. The operator preserves the type of the array or list. For instance, if you apply the `*.` operator to a `List`, the result is a `List`. All other expansion behavior is the same as for arrays.

See also

- “Enhancements on Gosu collections and related types” on page 189

Array flattening to a single dimensional array

If the property value on the original item returns an array of items, expansion behavior is slightly different. Gosu does not return an array of arrays, which is an array where every item is an array. Gosu returns an array containing all the individual elements of all the values in each array, which is known as *flattening* the array.

To demonstrate array flattening, create the following test Gosu class:

```
package test

class Family {
  var _members : String[] as Members
}
```

Next, paste the following lines into the Gosu Scratchpad:

```
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)
```

Running this program prints the following single-dimensional array:

```
["Peter", "Dave", "Scott", "Carson", "Gus", "Maureen"]
```

Examples

The following expression produces a name for each `Form` attached to a policy period.

```
PolicyPeriod.Forms*.DisplayName
```

The following expression returns a flattened array of display name values:

```
PolicyPeriod.ExposureUnits.Coverages*.DisplayName
```

Enhancements on Gosu collections and related types

Gosu collection and list classes rely on collection classes from the Java language. However, Gosu collections and lists have additional, built-in enhancements compared to Java. Gosu *enhancements* are Gosu methods and properties added to classes or other types without the subclassing that other languages require to make use of the new methods and properties.

See also

- “Enhancements” on page 201

Finding data in collections

Gosu provides the `firstWhere`, `where`, and `lastWhere` methods to enable you to find items in collections that match certain criteria. 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 as a parameter. In turn, this block takes an element as a parameter. The block has a Boolean expression in its body that uses the element and evaluates this expression to `true` or `false`. In the case that the Boolean expression evaluates to `true`, the block returns the element.

When you call the `where` method, the block executes once for each element in a collection of elements. In executing, the block evaluates the Boolean expression for each such element. The block returns all elements for which the

Boolean expression evaluates to `true`. In this way, the `where` method forms a new collection of elements. The method then returns this new collection. If the `where` method finds no matching items, it returns `null`.

The following example demonstrates how to find all items that match a criterion:

```
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 as a parameter. As in the case of the `where` method, this block takes an element as a parameter, evaluates a Boolean expression body for the element, and returns the element in the case that the expression evaluates to `true`. The `firstWhere` method returns the first element that the block returns. If the `firstWhere` method finds no matching items, it returns `null`.

The following example demonstrates how to find the first item that matches a criterion:

```
var strs = new ArrayList<String>(){"a", "ab", "abc"}
var firstLongerStr = strs.firstWhere( \ str -> str.length >= 2 )
```

The value of `firstLongerStr` is `"ab"` because `"ab"` is the first element in the list for which `str.length >= 2` evaluates to `true`.

The `lastWhere` method operates in a similar manner as the `where` and `firstWhere` methods. If the `lastWhere` method finds no matching items, it returns `null`.

Sorting lists or other comparable collections

Call the `sort` method on the list to sort a list in place.

```
myList.sort()
```

For descending order, use the `sortDescending` method.

By default, these methods do not use a localized sort collation order. For localized sorting, pass a comparator as a method argument.

Use `gw.api.util.LocaleUtil` to get the comparator for the current locale. For example:

```
myList.sort(LocaleUtil.getComparator())
```

If you need a comparator for a different locale, `LocaleUtil` provides methods to get a comparator by language or locale.

To retain the existing list and create a new list rather than sorting in place, use the `orderBy` and `orderByDescending` methods.

Sorting by a value other than the list element

The `sortBy` method on a list supports sorting on an element that you derive from the list element. For example, suppose you had an array list of `String` values that you want to sort by the text length. Create a block that takes a `String` and returns the sort key, which in this case is the number of characters of the parameter.

The `sortBy` method sorts the original list in place, and also returns a new list that contains the sorted values. To retain the existing list and return a new list, use the `orderBy` method.

The following example sorts strings by their length:

```
var myStrings = {"a", "abcd", "ab", "abc"}
var resortedStrings = myStrings.sortBy( \ str -> str.length )
```

The following line prints the contents of the collection of sorted strings:

```
resortedStrings.each( \ str -> print( str ) )
```

This line produces the output:

```
a
ab
abc
abcd
```

Similarly, you can use the `sortByDescending` method, which sorts in the opposite order.

For both of these methods, the block must return a comparable value. Comparable values include `Integer`, `String`, or any other values that can be compared with the greater than (`>`) or less than (`<`) operators.

By default, these methods do not use a localized sort collation order. For localized sorting, pass a comparator as a second argument to the method.

Advanced sort comparison

In some cases, comparison among your list objects is less straightforward. You might require more complex Gosu code to compare two items in the list. In such cases, use the sort method signature that takes a block that compares two list items.

The block must take two list elements and return `true` if the first element comes before the second, or otherwise return `false`. Using this method signature, the earlier sorting example looks like the following code:

```
var strs = new ArrayList<String>(){"a", "abc", "ab"}
var sortedStrs = strs.sort( \ str1, str2 -> str1.length < str2.length )
```

Although using the `sort` method is powerful, in most cases Gosu code is more readable if you use the `sortBy` or `sortByDescending` methods.

By default, these methods do not use a localized sort collation order. For localized sorting, pass a comparator as a second argument to the method.

Mapping data in collections

Use the `map` method to create an array list from a collection. A block expression transforms each element in the collection and stores the result as element in the new array list. For example, the following Gosu code declares and initializes an `ArrayList` of `String` objects. Then, it maps the lengths of the strings to a new array list.

```
// Declare and initialize an array list of strings.
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}

// Create a new array list with the lengths of the strings.
var lengthsOnly = myStrings.map(\ str -> str.length)
```

After the preceding code executes, the variable `lengthsOnly` is an array list, with elements 1, 1, 2, 2, 3, and 4.

Implicit type inference

In the preceding example, the `map` method takes a function that converts a `String` to its length. The code does not declare the type of the block argument `str` explicitly. At compile time, `str` is statically typed by inference as a `String` argument, from the declaration of the variable `myStrings`. That declaration uses the generics syntax `ArrayList<String>` to specify an `ArrayList` of `String` objects.

Similarly, the `lengthsOnly` variable is statically typed by inference, from the type returned by the `map` method. The block function passed to the `map` method returns an `Integer`, so the `map` method returns as an `ArrayList` of `Integer` objects.

Explicit type declaration

Although implicit type inference makes type declarations for you automatically, declare the types explicitly to increase the clarity of your code. For example, the following Gosu code declares the types for the block argument `str` and the type of object returned by the `map` method.

```
// Declare and initialize an array list of strings.
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}

// Create a new array list with the lengths of the strings.
var lengthsOnly : List<Integer> as ArrayList = myStrings.map(\ str:String -> str.length)
```

The type declaration for the argument to the block uses the declaration `:String`. The type declaration for the object that `map` method returns is more complex. The code declares that the `map` method returns an object that implements the `List` interface, with the type declaration `: List<Integer>`. The code downcasts the type of the returned object to an `ArrayList`, with the `as` key word.

Note: The downcast to `ArrayList` is for clarity only. By convention, the Gosu compiler assumes that a type declaration of `List<T>` is a declaration of `ArrayList<T>`.

See also

- "Generics" on page 221

Iterating across collections

Suppose you want to print each element in a list. You could use the list method `each`, which can be used instead of a traditional Gosu `for` loop. The following example Gosu code chains the `map` and `each` functions to create a new list and process each element in the new list in a single Gosu statement.

```
// Declare and initialize a list of strings.
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}

// Create a new list with the lengths of the strings and
// iterate the new list to print each element
myStrings.map(\ str -> str.length).each(\ len -> print(len)) // Two methods chained in one statement
```

Use the `each` method to perform repeated actions with the elements in a collection.

You can achieve concise code by chaining the `map` and `each` methods in a single Gosu statement. Sometimes, the conciseness of method chaining makes your code hard for others to read and interpret. In these cases, you might instead assign the return value of the `map` method to a variable, and then call the `each` method on the variable. For example:

```
// Declare and initialize a list of strings.
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}

// Create a new list with the lengths of the string.
var strLengths = myStrings.map(\ str -> str.length)

// Iterate the new list to print each element.
strLengths.each( \ len -> print( len ) )
```

Method chaining discards the new list after executing the statement that includes the `map` method. If you need to access the new list returned by the `map` method later in your code, assign the return value to a variable.

Partitioning collections

Partitioning a collection creates a new `java.util.Map` of key/value pairs from values in a simple list. Gosu provides the `partition` and `partitionUniquely` methods to partition collections. Each method has a single parameter that is a Gosu block, which takes an element from the original list as the block argument. The value returned by the block becomes the key for that element in the new map. Gosu uses type inference to statically type the `Map` objects that the methods return.

The partition method

For example, suppose you have the following list of `String` values.

```
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}
```

The following line of code partitions the preceding list into a `Map` that uses the lengths of the various `String` values as the keys to those value.

```
var lengthsToStringsMap = myStrings.partition( \ str:String -> str.length)
```

Because some of the values in the original list have the same lengths, the created map contains four keys and six values, which the following example map definition illustrates:

```
Map { 1 -> ["a", "b"], 2 -> ["bb", "ab"], 3 -> ["abc"], 4 -> ["abcd"] }
```

If you partition a list with the `partition` method, subsequent code must handle the case where a map expression resolves to a list instead of a single value.

The partitionUniquely method

If you are certain that the list contains unique values. use the `partitionUniquely` method. For example, suppose you have a list with the following `String` values. each with a different length than the others.

```
var myStrings = new ArrayList<String>(){"bb", "a", "abcd", "abc"}
```

The following line of code partitions the preceding list into a `Map` that uses the lengths of the various `String` values as the keys to those value.

```
var lengthsToStringsMap = myStrings.partitionUniquely( \ str:String -> str.length)
```

Because each value in the original list has a different length, the created map contains four keys and four values, which the following example map definition illustrates:

```
Map { 1 -> "a", 2 -> "bb", 3 -> "abc", 4 -> "abcd" }
```

If more than one list element has the same calculated value for its key, the `partitionUniquely` method throws an exception.

Example of partitioning a collection

The following example illustrates a typical case of partitioning a collection.

```
uses gw.api.database.Query
uses gw.api.database.DBFunction

// Query for new contacts from yesterday.
var contactQuery = Query.make(Contact).withLogSQL(true)
var contactResult =
        contactQuery.compare(DBFunction.DateFromTimestamp(contactQuery.getColumnRef("CreateTime")),
        Equals, java.util.Date.Yesterday ).select()

// Partition the result into a map by contact public ID.
var contactsById = contactResult.partitionUniquely( \ contact -> contact.PublicID )
```

The value of `contactsById` is a `Map` of contact `PublicID` values to contacts created yesterday. You can access a contact entity instance and its properties from the map by using the public ID as the key.

Converting lists, arrays, and sets

To convert arrays, lists, and sets to other types, use the following enhancement methods:

- Call the `toArray` method to convert a `Set` or a `List` to an `Array`.
- Call the `toList` method to convert a `Set` or an `Array` to a `List`.
- Call the `toSet` method to convert a `List` or an `Array` to a `Set`.
- Call the `join` method to combine the elements of an `Array`, a `List`, or a `Set` into a `String`, separated by a delimiter that you specify. For example, the following line combines all the items in the array into a `String`, separated by commas:

```
joinedString = array.join(",")
```

Converting an array or a list to a set removes duplicate values.

The following code shows the effects of various conversions:

```
// Declare and initialize an array list of strings.
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}

// Create a new array list with the lenghts of the strings.
var lengthsOnly : List<Integer> as ArrayList = myStrings.map(\ str:String -> str.length)
var lengthsSet = lengthsOnly.toSet()
var lengthsArray = lengthsOnly.toArray()
var lengthsStr = lengthsOnly.join("|")
print(typeof lengthsOnly + " " + lengthsOnly.size() + " " + lengthsOnly)
print(typeof lengthsSet + " " + lengthsSet.size() + " " + lengthsSet)
print(typeof lengthsArray + " " + lengthsArray.length)
print(lengthsStr)
```

This code prints the following lines:

```
java.util.ArrayList<java.lang.Object> 6 [1, 1, 2, 2, 3, 4]
java.util.HashSet<java.lang.Object> 4 [1, 2, 3, 4]
java.lang.Object[] 6
1|1|2|2|3|4
```

Flat mapping a series of collections or arrays

Suppose you have a collection where each object has a property that is an array or collection. You can use the `flatMap` method to create a single `List` of elements from all the elements on the inner collections. 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`.

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.

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 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 code 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>`.

See also

- "Gosu arrays and the expansion operator" on page 67

Sizes and length of collections and strings are equivalent

For collections and strings, `length` and `size` have the same meaning in Gosu. Gosu adds enhancements for the `Collection` and `String` classes to support both the `length` and `size` properties. The `length` property is deprecated on lists and sets. For these types, use the `size` property. The `size` property is deprecated on arrays. For array types, use the `length` property

List of enhancement methods on collections

The following tables list some of the Gosu enhancements to collections and lists. The letter `T` refers to the type of the collection. The syntax `<T>` comes from the language feature known as generics. For example, for the argument that is listed as `conditionBlock(T):Boolean`, the argument is a block. That block must take exactly one argument of the list's type (`T`) and return a `Boolean`. Similarly, where the letter `Q` occurs, this letter represents another type. The text at the beginning, which is `conditionBlock` in that example, is a parameter that is a block and its name describes the block's purpose.

Note: If a type letter such as `T` or `Q` appears more than once in arguments or a return result, the letter represents the same type in each usage.

Enhancement methods and properties on Iterable<T>

The following table lists the additional methods and properties for iterable objects, which are objects that implement `Iterable<T>`.

Method or property	Description
`Count` property	Returns the number of elements in the `Iterable`
`single()`	If there is only one element in the `Iterable`, returns that value. Otherwise throws an `IllegalStateException`.
`toCollection()`	Returns this iterable object if the object is already of type `Collection`. Otherwise, copies all values out of this `Iterable` into a new `Collection`.

Enhancement methods and properties on Collection<T>

Most collection methods are implemented directly on `Collection`, not on `List` or other types that extend `Collection`. The following tables list the available methods and properties.

The following methods and properties provide information about the collection.

Method or property	Description
`allMatch(cond)`	Returns true if all elements in the `Collection` satisfy the condition.
`countWhere(cond)`	Returns the number of elements in the `Collection` that match the given condition.
`hasMatch(cond)`	Returns true if this `Collection` has any elements in it that match the given block.
`HasElements` property	Returns `true` if this `Collection` has any elements. Use this method rather than the default collection method `empty()` because `HasElements` interacts better with `null` values. For example, the expression `col.HasElements()` returns a non-true value even if the expression `col` is `null`.

The following methods provide functionality to perform an action on every item in a collection.

Method	Description
`each()`	Iterates each element of the `Collection`.
`eachWithIndex()`	Iterates each element of the `Collection` with an index.

The following methods provide functionality to get a particular element or value from a collection.

Method	Description
`average(selector)`	Returns the average of the numeric values selected from the `Collection<T>`.
`first()`	Returns the first element in the `Collection`, or return `null` if the collection is empty.
`firstWhere(cond)`	Returns the first element in the `Collection` that satisfies the condition, or returns `null` if none do.
`last()`	Returns the last element in the `Collection` or return `null` if the list is empty.
`lastWhere(cond)`	Returns the last element in the `Collection` that matches the given condition, or `null` if no elements match it.
`max()`	Returns the maximum of the selected values from `Collection<T>`. This method excludes any null values in the collection. If there are no non-null values, this method throws an exception.
`maxBy(proj)`	Returns the maximum `T` of the `Collection` based on the projection to a `Comparable` object.
`min()`	Returns the minimum of the selected values from `Collection<T>`. This method excludes any null values in the collection. If there are no non-null values, this method throws an exception.
`minBy(proj)`	Returns the minimum `T` of the `Collection` based on the projection to a `Comparable` object.
`singleWhere(cond)`	If there is only one element in the `Collection` that matches the given condition, returns that value. Otherwise, throws an `IllegalStateException`.
`sum(proj)`	Returns the sum of the numeric values selected from the `Collection<T>`.

The following methods provide functionality to sort the elements in a collection.

Note: By default, these methods do not use a localized sort collation order. For localized sorting, pass a comparator as a second argument to the method.

Method	Description
`orderBy(proj)`	Returns a new `List<T>` ordered by a block that you provide. This method is different from `sortBy`, which is retained on `List<T>` and which sorts in place.
`orderByDescending(proj)`	Returns a new `List<T>` reverse ordered by the given value. This method is different from `sortByDescending`, which is retained on `List<T>` and which sorts in place.
`reverse()`	Reverses the collection as a `List`.
`thenBy(proj)`	Additionally orders a `List` that has already been ordered by `orderBy`.
`thenByDescending(proj)`	Additionally reverse orders a `List` that has already been ordered by `orderBy`.

The following methods provide functionality to filter the elements in a collection.

Method	Description
`removeWhere(cond)`	Removes all elements that satisfy the given criteria.
`retainWhere(cond)`	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.

Method	Description
`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.

The following methods provide functionality to convert the collection to another class or format.

Method	Description
`asIterable()`	Returns this `Collection<T>` as a pure `Iterable<T>`, not as a `List<T>`).
`flatMap(proj)`	Maps each element of the `Collection` to a `Collection` of values and then flattens them into a single `List`.
`fold()`	Accumulates the values of an `Collection<T>` into a single `T`.
`join()`	Joins all elements together as a string with a delimiter.
`map(proj)`	Returns a `List` of each element of the `Collection<T>` mapped to a new value.
`partition(proj)`	Partitions this `Collection` into a `Map` of keys to a list of elements in this `Collection`.
`partitionUniquely(proj)`	Partitions this `Collection` into a `Map` of keys to elements in this `Collection`. Throws an `IllegalStateException` if more than one element maps to the same key.
`reduce(init, reducer)`	Accumulates the values of a `Collection<T>` into a single `V` given an initial seed value.
`toList()`	If this `Collection` is already a list, returns the same `Collection`. Otherwise creates a new `List` and copies this `Collection` to it.
`toSet()`	Converts the `Collection` to a `Set`.
`toTypedArray()`	Converts this `Collection<T>` into an array `T[]`.

The following methods provide functionality to act on two collections.

Method	Description
`disjunction(coll)`	Returns a new `Set<T>` that is the set disjunction of this collection and the other collection.
`intersect(iter)`	Returns a `Set<T>` that is the intersection of the two collection objects.
`subtract(coll)`	Returns a new `Set<T>` that is the set subtraction of the other collection from this collection.
`union(coll)`	Returns a new `Set<T>` that is the union of the two collections.

Enhancement methods on List<T>

The following table lists the available methods on `List<T>`.

Method	Description
`copy()`	Creates a copy of the list.
`freeze()`	Returns a new unmodifiable version of the list.
`reverse()`	Reverses the `Iterable`.
`shuffle()`	Shuffles the list in place.
`sort()`	Sorts the list in place, with optional support for localized sorting. By default, this method does not use a localized sort collation order. For localized sorting, pass a comparator as a second argument to the method.

Method	Description
`sortBy(proj)`	Sorts the list in place but instead of comparing the items directly like sort does, uses a block that returns the item to compare. By default, this method does not use a localized sort collation order. For localized sorting, pass a comparator as a second argument to the method.
`sortByDescending()`	Same as `sortBy`, but sorts the list in place in descending order.
`sortDescending()`	Same as `sort`, but sorts the list in place in descending order.

Enhancement methods and properties on Map<K,V>

The following tables list the available methods and properties on `Map<K,V>`. The symbol `K` represents the key class. The symbol `V` represents the value class.

The following methods and properties provide functionality for general usage of a map.

Method or property	Description
`copy()`	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.
`Count` property	The number of keys in the map.
`Keys` property	A collection of all keys in the map.
`toAutoMap(defaultValueBlock)`	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.
`Values` property	A collection of all values in the map.

The following methods provide functionality to perform an action on every item in a map.

Method	Description
`eachKey(block)`	Runs a block for every key in the map. The block must take one argument (the key) and return no value.
`eachKeyAndValue(block)`	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.
`eachValue(block)`	Runs a block for every value in the map. The block must take one argument (the value) and return no value.

The following methods provide functionality to filter and remap a map. 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	Description
`filterByKeys(keyFilter)`	Returns a new map that is a clone of the original map but without entries whose keys do not satisfy the *`keyFilter`* expression. The key filter block must take one argument (a key) and return `true` or `false`.
`filterByValues(valueFilter)`	Returns a new map that is a clone of the original map but without entries whose values do not satisfy the *`valueFilter`* expression. The key filter block must take one argument (a value) and return `true` or `false`.
`mapValues(mapperBlock)`	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.

Method	Description

	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 `Map<Integer,String>` and maps it to a new object of type `Map<Integer,Integer>`:
	`var origMap = { 1 -> "hello", 4 -> "there"}`
	`var newMap = origMap.mapValues( \ value -> value.length )`

`retainWhereKeys(keyFilter)`	Destructively removes all entries whose keys do not satisfy the *`keyFilter`* 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`.
`retainWhereValues(valueFilter)`	Destructively removes all entries whose values do not satisfy the *`valueFilter`* 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`.

The following methods provide functionality to use properties files for a map.

Method	Description
`readFromPropertiesFile(file)`	Reads the contents of this map from a file in the Java properties format using APIs on `java.util.Properties`. The method takes a `java.io.File` object and returns a map of the keys and values. The return type is `Map<String,String>`.
`writeToPropertiesFile(file)` `writeToPropertiesFile(file, comments)`	Writes the contents of this map to a file in the Java properties format using APIs on `java.util.Properties`. The method takes a `java.io.File` object and returns nothing. The second method signature takes a `String` object to write comments in the properties file.

Enhancement methods on Set<T>

The following table lists the available methods on `Set<T>`.

Method	Description
`copy()`	Creates a copy of the set
`freeze()`	Returns a new unmodifiable version of the set
`powerSet()`	Returns the power set of the set

See also

- “Enhancements” on page 201
- “Generics” on page 221
- “Sorting lists or other comparable collections” on page 190
- “Wrapped maps with default values” on page 186
- “Flat mapping a series of collections or arrays” on page 194

chapter 13

Enhancements

Gosu enhancements allow 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.

Enhancements differ from subclasses in important ways. Enhancement methods and properties on a class are available to all objects of the enhanced class. Methods and enhancements on subclass are available only to objects of the subtype, not to objects of the superclass. Use enhancements to add powerful functionality omitted by the original authors.

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 feature is most useful if the source code for a class is unavailable or class is marked final and so cannot be subclassed. Enhancements can be applied to interfaces as well as classes, which enables you to 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. 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 provide 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. An enhancement does not add new methods to the interface itself, nor add new requirements for classes that implement the interface. Instead, enhancing an interface provides all objects whose class implements the interface with new methods and properties. For example, if you enhance the `java.util.Collection` interface with a new method, all collection types 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.

Static dispatch of enhancements

Gosu enhancements are statically dispatched. Enhancement code is run at run time, but availability and the relevant code is determined at compile time. Availability of enhancement properties and methods is based on the compile-time type, not the run-time type. This behavior is different from regular properties and methods that are accessed at run time.

This distinction is important for two reasons:

- **Subtype/supertype distinctions might prevent access** – Suppose a method argument is declared as the type `Object`. At compile time, the argument variable has type Object, even though at run time it could be a much more specific type than `Object`. If at run time the object is `java.util.ArrayList`, the enhancement method on `ArrayList` is unavailable. You can work around this limitation by downcasting (`myObject as ArrayList`) before accessing the enhancement. However, downcasting could fail at run time if the run-time type is not compatible with `ArrayList`, so you must understand the risks of that technique.
- **Subtype/supertype distinctions might dispatch in unexpected ways** – If you have many different enhancements with the same names for properties and methods, there could be enhancements on more than one type in the hierarchy. For example, if you enhanced `java.util.Map` interface and `java.util.HashMap` with different enhancement logic, the one that is used is the most specific type at compile time for that programming context. If you use a variable declared to type `java.util.Map`, its enhancement methods are determined at compile time, even if at run time that object has subtype `java.util.HashMap`.

See also

- "Blocks" on page 175
- "Collections" on page 183

Syntax for using enhancements

Using an enhancement requires no special syntax. 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, suppose you are typing "`s1.calculateHash()`". In typing this statement, 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 a package folder, and then click **New→Gosu Enhancement**. In the dialog that appears, you can enter the enhancement class name and the type, which is typically a class name or entity type name, to enhance. Studio creates a new enhancement with the appropriate syntax.

Syntax for defining enhancements

Although using enhanced properties and methods is straightforward, a special syntax is necessary for defining new enhancements. Defining a new Gosu enhancement looks similar to defining a new Gosu class, with some minor differences in their basic definition.

Differences between classes and enhancements include:

- Use the keyword `enhancement` instead of `class`.
- To define what to enhance, use the syntax: "`: TYPETOEXTEND`" instead of "`extends CLASSTOEXTEND`".
- To reference methods on the enhanced class/type, use the symbol `this` to access the enhancements. 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 `super` 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, these new properties 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 print 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 enhancement in the Gosu Scratchpad 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 prints:

```
[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 use the convention of the enhanced type name, then an optional functional description, and finally the word `Enhancement`, as shown in the following line:

```
[EnhancedTypeName][OptionalFunctionalDescripton]Enhancement
```

For example, to enhance the `Report` class, you could call your enhancement:

```
ReportEnhancement
```

If the enhancement adds methods related to financials data, you can 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 extremely rare cases, you might need to enhance a built-in type and to use a `protected` property or method. If so, you might need to define your enhancement in a subpackage of the enhanced type. However, to avoid namespace conflicts with built-in types, avoid this approach if possible.

See also

- “Modifiers” on page 157
- “Naming conventions for packages and types” on page 150

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 means all arrays.

part 2

Advanced Gosu features

chapter 14

Annotations

Gosu annotations are a syntax to provide metadata about a Gosu class, constructor, method, property, field, or parameter. An annotation can control the behavior of the type, the documentation for the type, or the behavior of the code editor.

Annotating a class, method, type, class variable, or argument

Gosu annotations are a syntax to provide metadata about a Gosu class, constructor, method, property, field, or parameter. An annotation can control the behavior of the type, the documentation for the type, or the behavior of the code editor.

For example, annotations can indicate what a method returns or 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 the item to annotate.

For example, the following example declares that a class is deprecated:

```
@Deprecated
class MyServiceAPI {
  public function myRemoteMethod() {}
}
```

You can use both annotations that are defined natively in Gosu and also Java annotations.

In your annotation usage, you can use either the fully qualified name of the annotation class or just the name. If you do not use the fully qualified name, add a `uses` line for that annotation class at the top of the file. For example, the following line shows the `uses` line for the `@Deprecated` annotation:

```
uses java.lang.Deprecated
```

In some cases, you follow the annotation name with an argument list within parentheses. The following example uses arguments to the annotation to specify that a function might throw a specific exception:

```
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 you do not provide arguments to an annotation, you can omit the parentheses. For example, suppose you add an annotation called `MyAnnotation` that takes no arguments. You could use `MyAnnotation` in the following, verbose syntax:

```
@MyAnnotation()
```

Because there are no arguments, you can optionally omit the parentheses:

```
@MyAnnotation
```

Gosu requires argument lists to be in the same format as regular function or method argument lists:

```
// Standard Gosu argument lists
@Param("str", "The String value to parse")
```

Gosu annotations optionally support the named arguments calling convention, which includes a colon before the argument name, and commas between arguments:

```
@Param(:FieldName = "str", :FieldDescription = "The String value to parse")
```

See also

- “Named arguments and argument defaults” on page 112
- “Importing types and package namespaces” on page 114

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 the 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
  }
}
```

The Gosu compiler permits argument annotations. However, Gosu does not support special compiler or IDE behavior for Java 8 Type annotations such as `@m`, `@Nonnull`, `@ReadOnly`, `@Regex`, `@Tainted`, and `@Untainted`.

See also

- “Importing types and package namespaces” on page 114

Built-in annotations

The Gosu language includes built-in annotations defined in the `gw.lang.*` package. This package is always in scope, so the fully qualified annotation 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 for any specific 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 the circumstances that cause the exception to be thrown, 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 `String` warning 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.	1. A `String` value to indicate the 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, on the line before the class declaration, add: `@SuppressWarnings("deprecation")`

Internal annotations

Some Gosu annotations are reserved for internal use. You may see internal annotations in built-in Gosu code. These annotations are not supported for your use. The following table lists these internal annotations so that you understand their role in built-in classes.

Internal annotation	Description	Usage limits
`@Export`	Let a class be visible and editable in Studio	Classes only
`@ReadOnly`	Let a class be visible in Studio but non-editable. Studio does not permit copying a read-only class into the `configuration` module for modification.	Classes only

No support for Java 8 type annotations

The Gosu compiler permits argument annotations. However, Gosu does not support special compiler or IDE behavior for the Java 8 Type annotations such as `@m`, `@Nonnull`, `@ReadOnly`, `@Regex`, `@Tainted`, and `@Untainted`.

Web service annotations

Several built-in annotations related to publishing web services are available.

Examples

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 component is temporarily available, but a future release will remove that component. Immediately start to refactor code that uses deprecated APIs. This refactoring ensures that your code is compatible with future releases. Following this advice simplifies future upgrades.

```
package example

class MyClass {

  @Deprecated("Don't use MyClass.myMethod(). Instead, use betterMethod().")
  public function myMethod() {print("Hello")}

  public function betterMethod() {print("Hello, World!")}
}
```

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 as:

```
@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 {
  ...
}
```

See also

- *Integration Guide*

Getting annotations at run time

You can get annotation information from a class either directly or by getting the type from an object at run time. You can get the type of an object at run time by using the `typeof` operator, such as: `typeof` *`TYPE`*.

Annotation access at run time

You can get annotation information from a type, a constructor, a method, or a property by accessing the metadata information objects that are attached to the type.

To get a single instance of a specific annotation, call the `getAnnotation` method on the type.

For a list of multiple annotation instances of one type, call the `getAnnotationsOfType` of the type.

To get an annotation from targets such as constructors, methods, or properties, call the `getAnnotation` method on the item, not on the type itself. The following table shows how to get an annotation from each target type. In the examples in the table, the variable `i` represents the index in the list. In practice, you would probably search for the required target by name using `List` methods like `list.firstWhere(\ s -> s.Name = "MethodName")`.

Get annotations on this object	Example Gosu to access the example `@MyAnnotation` annotation
Type	`(typeof obj).TypeInfo.getAnnotation(MyAnnotation)`
Constructor for a type	`(typeof obj).TypeInfo.Constructors[i].getAnnotation(MyAnnotation)`
Method of a type	`(typeof obj).TypeInfo.Methods[i].getAnnotation(MyAnnotation)`
Property of a type	`(typeof obj).TypeInfo.Properties[i].getAnnotation(MyAnnotation)`

Getting multiple annotations of multiple types

You can get all annotations of any annotation type using the two properties `Annotations` and `DeclaredAnnotations`. These two properties resemble the Java versions of annotations of the same names.

On types and interfaces, `Annotations` returns all annotations on this type or interface and on all its supertypes or superinterfaces. `DeclaredAnnotations` returns annotations only on the given types, ignoring supertypes or 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 the required target by name using `List` methods like `list.firstWhere(\ s -> s.Name = "MethodName")`.

Get all annotations on...	Example
Type	`(typeof obj).TypeInfo.Annotations`
Constructor	`(typeof obj).TypeInfo.Constructors[i].Annotations`
Method	`(typeof obj).TypeInfo.Methods[i].Annotations`
Property	`(typeof obj).TypeInfo.Properties[i].Annotations`

The return result of these methods is typed as a list of the expected type. Using the examples in the previous table, the result is of type `List<MyAnnotation>`. This type is shown using generics syntax, which means “a list of instances of the `MyAnnotation` annotation class”.

See also

- “Getting annotations at run time” on page 212
- “Generics” on page 221

Getting annotation data from a type at run time

You perform the following general tasks to get annotation data from a type at run time.

1. Get a reference to a type. If you have a reference to an object instead of its type, get the type at run time using the `typeof` operator.
2. Get the result’s `TypeInfo` property.
3. To access annotations on the type, call the `getAnnotation` method, passing your annotation class name directly as a parameter. If you want to get annotations from the constructor, method, or property, get the subobject as described in the previous table, then call its `getAnnotation` method.
4. On the result, get its `Instance` property or call its `getInstance` method. The `Instance` property throws an exception if there is more than one instance of that annotation on the type in that context.

5. Cast the result to the desired annotation type using the `as` keyword. For example, suppose the result of the previous step is a variable called `res` and your annotation is called `@MyAnnotation`. Use the syntax: `as MyAnnotation`.
6. From the result, get the data from the annotation instance with any annotation arguments defined as methods. For example, if an annotation has constructor arguments `purpose` and `importance`, get those values using methods `purpose` and `importance`.

Example

The classes `example.MyClass` and `example.MyAnnotation` demonstrate the prior steps:

```
package example

annotation MyAnnotation {
  function purpose() : String
  function importance() : int = 0
}

package example

@MyAnnotation(:purpose="to do the right thing", :importance=44)
class MyClass {
}
```

The following code uses the `example.MyClass` and `example.MyAnnotation` classes:

```
uses example.MyAnnotation
uses example.MyClass

// Get a reference to an object of that class
var c = new MyClass()

// Get the type info
var ti = (typeof c).TypeInfo

// Get the annotation of our specific annotation type
var ann = ti.getAnnotation(MyAnnotation).Instance as MyAnnotation

print("Purpose is " + ann.purpose())
print("Importance is " + ann.importance())
```

This example prints the following lines:

```
Purpose is to do the right thing
Importance is 44
```

Defining your own annotations

You can define annotations to add entirely new metadata annotations, apply them to programming declarations, and optionally retrieve 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.

You can define new annotation types that can be used in other Gosu code. Annotations are defined in `.gs` files like Gosu classes, but they use a different syntax for the declaration. Your annotation can take arguments, which can set metadata that can be queried by IDEs or dynamically at run time.

The declaration syntax:

- Instead of using the `class` keyword, use the `annotation` keyword.
- Define each annotation argument as if it were a class method, ensuring you declare the return type
- To specify a default value to an annotation argument, append an equal sign and then the default value.

For example, the following code defines an annotation with one `String` argument and one integer primitive, with a default value for the integer:

```
package example

annotation MyAnnotation {
  function purpose() : String
  function importance() : int = 0
}
```

You can now use this annotation on a new class:

```
package example

@MyAnnotation(:purpose="do the right thing", :importance=44)
class MyClass {
}
```

By default, this annotation can be used on any method, type, property, or constructor, and as many times as necessary.

See also

- "Getting annotations at run time" on page 212

Customizing annotation context and behavior

Gosu supports special annotations on an annotation declaration to customize its behavior. Annotations on an annotation declaration are called *meta-annotations*.

Limiting annotation usage to specific code contexts

Usage of each annotation can be customized, such as allowing the annotation only under certain conditions. For example, the built-in annotation `@Returns` can appear only on methods, not on classes. To restrict usage in this way, use the native Java `java.lang.annotation.Target` annotation on the declaration of your annotation and pass a list of targets. The *target* defines where the annotation can be used.

Declare targets with a list of instances of the `java.lang.annotation.ElementType` enumeration:

- `METHOD` – This annotation can be used on a method. Gosu properties are dynamic and similar to methods, so use this target for Gosu properties.
- `TYPE` – This annotation can be used on a type, including classes
- `FIELD` – This annotation can be used on a field or enumeration constant. Because Gosu properties are dynamic and similar to methods, instead use `METHOD` for Gosu properties.
- `CONSTRUCTOR` – This annotation can be used on a constructor
- `PARAMETER` – This annotation can be used on a function argument (a parameter)
- `ANNOTATION_TYPE` – This annotation can be used on annotation type declarations

Unlike Java, Gosu does not support the value `LOCAL_VARIABLE` for annotations on local variable declarations.

If an annotation uses no `@Target` annotation, the Gosu default is to allow the annotation on all parts of a type.

You can use the compact Gosu syntax to define lists of items by surrounding the values with braces (`{}`). Additionally, you can omit `ElementType.` before the static values for the enumerations.

For example, the following annotation declaration defines valid usage to be types, including Gosu classes, and methods:

```
package example

uses java.lang.annotation.ElementType
uses java.lang.annotation.Target

@Target({ METHOD, TYPE })
annotation MyAnnotation {
  function purpose() : String
  function importance() : int = 0
}
```

Setting retention of an annotation

Gosu supports the native Java meta-annotation `@Retention`. Pass one of the following `RetentionPolicy` enumeration constants:

- `CLASS` – Annotations are recorded in the class file but need not be retained by the VM at run time.
- `RUNTIME` – Annotations are recorded in the class file and retained by the VM at run time. These annotations can be read reflectively.
- `SOURCE` – Annotations are discarded by the compiler.

For example:

```
uses java.lang.annotation.Retention

@Retention( RUNTIME)
annotation MyAnnotation {
  function purpose() : String
  function importance() : int = 0
}
```

Setting inheritance of an annotation

Gosu supports the native Java meta-annotation `@Inherited`. It takes no arguments. Use it to indicate that this annotation is inherited by subtypes.

Setting documented status of an annotation

Gosu supports the native Java meta-annotation `@Documented`. It takes no arguments. This annotation indicates that an annotation is to be documented by automatic documentation tools.

chapter 15

Dimensions

Dimensions represent a quantifiable amount that can be measured in one or more units.

Dimensions overview

Dimensions represent a quantifiable amount that can be measured in one or more units. Dimensions can include any measurable physical or abstract dimension, for example:

- Length, such as 1 mm
- Time, such as 1 day or 1 year
- Weight, such as 1 kg
- Money, such as 1 US Dollar
- Amount of computer storage space, such as 1 GB

Dimensions can reference complex combinations of dimensions. Speed can be represented by a unit that combines distance and time, such as 60 miles per hour. Acceleration can be represented as speed per time unit, such as 1 meter per second per second.

Each Gosu dimension is represented by a class that implements the interface `gw.lang.IDimension`. You create custom dimensions by implementing the `IDimension` interface. In your class declaration, parameterize the interface as follows:

```
final class YOURCLASSNAME implements IDimension<YOURCLASSNAME, UNITNAME>
```

Replace *`YOURCLASSNAME`* with your class name. Replace *`UNITNAME`* with a unit. The unit type might be a built-in numeric type like `int` or `BigDecimal`, but could also be any arbitrary type including custom types.

The default syntax for creating a dimension object is the standard `new` operator with constructor arguments. A typical dimension class uses at least one numeric constructor argument and optionally a unit type if the context requires the unit type. For example, a new monetary amount object representing 3 US Dollars might require a numeric value (3) and the unit (US Dollars):

```
var xx = new MonetaryAmount(3, Currency.TC_USD)
```

Create as many constructors as typical usage of your custom dimension type requires.

Gosu natively supports basic arithmetic functions on dimensions as part of the `IDimension` interface. The Gosu parser looks for special methods for each operator and checks the argument types to see what types of operations are supported. For example, a `Velocity` dimension can define a `multiply` method that takes a `Time` argument and returns a `Length` dimension. Gosu uses this information to let you use the `*` operator to multiply a `Velocity` and `Time` dimension, which generates a `Length` object:

```
// Multiple Velocity and Time to get a Length object representing 150 miles
var distance = velocity * time
```

The following methods implement arithmetic on a custom dimension. The argument must be the type of the object on the right hand side of the arithmetic operator. The return type can be any type.

- `add` – Addition using the `+` operator
- `subtract` – Subtraction using the `-` operator
- `multiply` – Multiplication using the `*` operator
- `division` – Division using the `/` operator
- `modulo` – Modulo (remainder after division) using the `%` operator

Additionally, you can support the unary negation operator (`-`) by implementing the `negate` method, which takes no arguments.

Dimensions expressions example (Score)

The following example creates a custom dimension called `Score` that represents a score in a game. The score measures progress for each player in a point value stored as a `BigDecimal` value.

The following is the `Score` class:

```
package example

uses java.math.BigDecimal

final class Score implements IDimension<Score, BigDecimal> {

  // Declare a Points property
  var _points : BigDecimal as Points

  /* REQUIRED BY IDimension */
  construct( p : BigDecimal) {
    _points = p
  }

  construct() {
    _points = 0 // If used with a no argument constructor, just set points to 0
  }

  override function toNumber(): BigDecimal {
    return _points
  }

  override function fromNumber(bigDecimal: BigDecimal): Score {
    return new Score(bigDecimal)
  }

  override function numberType(): Class<BigDecimal> {
    return BigDecimal
  }

  /* REQUIRED BY Comparable */
  override function compareTo(o: Score): int {
    return (_points.compareTo(o.Points))
  }

  /* Improve printing... */
  public function toString() : String {
    return "Score is " + _points + " points"
  }

  // ADDITIONAL conversions

  function fromNumber(i: int): Score {
    return new Score(i as BigDecimal)
  }

  function fromNumber(i: Integer): Score {
    return new Score(i as BigDecimal)
  }
```

```
/* ENABLE arithmetic for Dimensions */

  function add( td: Score) : Score {
    return new Score(td.Points + _points)
  }

  function subtract( td: Score) : Score {
    return new Score(td.Points + _points)
  }

  function multiply( bd : BigDecimal) : Score {
    return new Score(bd * _points)
  }

  function multiply( td : Score) : Score {
    return new Score(td.Points * _points)
  }

  function divide( bd : BigDecimal) : Score {
    return new Score(_points / bd)
  }

  function divide( td: Score) : Score {
    return new Score(_points / td.Points)
  }
}
```

Consider the following code that you can run in the Gosu Scratchpad:

```
uses example.Score

var x = new Score()
var y = new Score()
var addme = new Score(2)

print ("x = " + x + " ..... y = " + y)

x = x + addme
y = y + addme + addme
print ("x = " + x + " ..... y = " + y)

var z = x * y.Points
print ("z = " + z )
```

This example prints the following:

```
x = Score is 0 points ..... y = Score is 0 points
x = Score is 2 points ..... y = Score is 4 points
z = Score is 8 points
```

Built-in dimensions

The built-in dimensions are as follows:

- `gw.pl.currency.MonetaryAmount`
- `gw.api.financials.CurrencyAmount`

Both classes have a numeric component and a currency from the `Currency` typelist. For example, you can instantiate each of them as follows:

```
var ma = new MonetaryAmount(3, Currency.TC_USD)
var ca = new CurrencyAmount(3, Currency.TC_USD)
```

chapter 16

Generics

Generics is a language feature that you use to define a class or function that applies to many types by abstracting its behavior across multiple types of objects. Gosu generics work in a very similar manner to generics in Java.

You use generics to write statically typed code that can be abstracted to work with multiple types. By designing an API to apply to different types of objects, you write the code only once, and the code supports those different types. In this way, you can generalize classes or methods to work with multiple types and retain compile-time type safety.

Generics make the relationships explicit between the types of parameters and return values of a function. Generics are especially useful if any of the parameters or return value are collections. For example, you can require two arguments to a function to be homogeneous collections of the same type of object, and that the function returns the same type of collection. Designing functions to be abstract in that way enables your code and the Gosu language to infer other relationships. For example, a function that returns the first item in a collection of `String` objects is always typed as a `String`. You do not need to write coercion code with the syntax `as` *`TYPE`* with functions that use generics. Because generics increase how often Gosu can use type inference, your collection-related code can be readable, concise, and type-safe.

See also

- "More about the Gosu type system" on page 38

Overview of Gosu generics

The greatest power of Gosu generics is defining an API that works with multiple types, not just a single type. To specify that an item is not restricted to a single type requires a syntax called *parameterization*. Functions defined with generics clarify the relationships among the types of parameters and return values. For example, even though the type of a parameter is generic, the return value can be strongly typed to match the exact type of that parameter. Generics are particularly useful in specifying the type of object that a collection parameter or return value contains. For example, the method returns a "list of `MyClass` objects" or "a `MyClass` object", rather than a "list of `Object` objects" or just "an `Object` object".

If a method uses Gosu generics to define arguments or a return value, the API is straightforward to use for an API consumer. The syntax for defining the type of item is angle bracket notation such as *`CLASS<USING_TYPE>`*, *`FUNCTION<USING_TYPE>`*, or *`COLLECTION_CLASS<OF_TYPE>`*. The parameterized definition of a method or class uses a generics wildcard inside the angle brackets. By convention, the wildcard is `T`, so the generics syntax is `<T>`. To use a collection that is an argument to or return value from a method that uses generics, you define the collection as a specific type. For example, the following line defines a list of `String` objects, which you can use as an argument to a generic method.

```
var myStrs = new ArrayList<String>() { address1, address2, address3, address4 }
```

Note: In practice, you sometimes do not need to define the type because of type inference or special object constructors. For example, `var myStrs = { "a", "ab", "abc"}` is equivalent to `var myStrs = new ArrayList<String>() { "a", "ab", "abc"}`.

You can create a method that parameterizes an argument as a specific type of list, in this case a list of strings. This argument has an explicit type and does not use generics to support multiple types of array list. The code in the method uses methods that are specific to the `String` type, so does not support other list types.

```
function printStrings( strs : ArrayList<String> ) {
  for ( s in strs ) {
    print( "${s} ${s.length}" )
  }
}
```

To call a method that uses a parameterized argument type, call the method in the usual way:

```
printStrings( myStrs )
```

A compile-time error occurs if you use an argument that does not match the parameterized specification on a method that uses Gosu generics to specify the argument type:

```
printStrings( {1, 2, 3, 4} ) // Array list of Integer is not an array list of String
```

Parameterizing collections and other containers is a typical use of generics, but Gosu does not limit generics support to only container types. You can define any class to use Gosu generics to generalize what the class supports or how the class works with various types.

Gosu provides two ways to use generic types:

- You can *parameterize a class*, which adds generic types to a class definition.
- You can *parameterize a method*, which adds generic types to a method definition.

Compatibility of Gosu generics with Java generics

Gosu generics are compatible with generics implemented in Java version 1.8. You can use Java utility classes designed for Java 1.8 generics and even extend them in Gosu.

No support for super syntax in Gosu generics

There is one exception for Java-Gosu compatibility, which is that Gosu does not support the syntax `<? super TYPE>`.

Reification of Gosu generics

One important difference between Gosu and Java is that Gosu generics are reified. *Reified* Gosu generics retain the actual specific type at run time. For example, at run time, you can check whether an object is an instance of `PetGroup<Cat>` or `PetGroup<Dog>` including the information in the angle brackets.

Java generics have *type erasure*, in which generics lose the generic parameter information at run time. Java generics maximize compatibility with older Java code that did not support generics.

Gosu does not reify the type of a generic Java object. If you want the reification behavior for a Java class that Gosu provides for Gosu classes, wrap the Java class in a Gosu class.

Example of reifying a Java type

The following Gosu class uses a Java `HashMap` object. This object is not reified.

```
class MyTester {
  // Return a map from two lists
  // This signature returns a Java object that has its type erased
  public reified function mapArray<K,V>(kArray : ArrayList<K>, vArray : ArrayList<V>) : HashMap<K,V> {

  // This signature returns a reified Gosu object
```

```
  // public reified function mapArray<K,V>(kArray : ArrayList<K>, vArray : ArrayList<V>) : MyMapper<K,V> {

    var theMap : HashMap<K,V>
    for (v in vArray index i) {
      theMap.put(kArray[i], v)
    }
    return theMap
  }
}
```

In the `mapArray` method, the symbols `K` and `V` are used as types and Gosu matches `K` and `V` to the types of the collections passed into the method.

Code can use this class:

```
var myKeys = new ArrayList<Integer>(){ 1, 2, 3, 4}
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "123"}
var t = new MyTester()
var myMap = t.mapArray(myKeys, myStrings)

print("My map is a ${typeof  myMap}")
```

The variable `myMap` is typed as `Hashmap<java.lang.Object, java.lang.Object>`.

To cause reification of the `myMap` variable, define a Gosu class that wraps `HashMap`.

```
class MyMapper<K,V> extends HashMap {

}
```

Comment out the method signature in the `MyTester` class that returns a `HashMap` object and uncomment the signature that returns a `MyMapper` object. In the code that uses the `MyTester` class, the variable `myMap` is now typed as `MyMapper<java.lang.Integer, java.lang.String>`.

See also

- “Restricting generics” on page 226

Parameterizing a class

To specify a class that operates with a generic type, define the class with the angle bracket notation `CLASSNAME<GENERIC_TYPE_WILDCARD>`. By convention, for `GENERIC_TYPE_WILDCARD`, use a one-letter name, 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:

```
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]
  }
}
```

Other code can use this class and pass an array list of any type to the method:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}

var t = new GenericsTest<String>()
var first = t.printAndReturnFirst( myStrings )
```

The variable `first` is strongly typed as `String` at compile time because the code uses a method that was defined with generics.

Because the code is strongly typed at compile time, the reliability of the code improves at run time.

Instantiating a generic class

Enhancements of generic classes

If you create an enhancement of a generic class, you must use the correct syntax:

- For a generic enhancement, you parameterize both the enhancement name and the class name. For example, the declaration of a generic enhancement to the `GenericsTest` class looks like the following line.

```
enhancement GenericsTest_Enh<T> : GenericsTest<T>
```

- For an enhancement that applies to a specific type on the generic class, you specify the type only on the class name. For example, the declaration of an enhancement to the `String` type of the `GenericsTest` class looks like the following line.

```
enhancement GenericsTest_StringEnh : GenericsTest<String>
```

See also

- "Enhancements" on page 201

Parameterizing a method

You can add a finer granularity of type usage by adding generic type modifiers to a method, immediately after the method name. In Gosu, this syntax of adding the generic type modifiers is called *parameterizing* the method. In other languages, this syntax of adding the generic type modifiers is known as making a *polymorphic method with a generic type*.

If the parameterized method needs information about the runtime type of the type parameter, you must add the modifier `reified` to the method declaration. Some parameterized methods do not require information about the runtime type of the type parameter. For these methods, the modifier `reified` is optional.

For example, in the following code, the class is not parameterized but one method is parameterized and does not require information about the type of `T` at run time:

```
class GenericTypesTester<T> {
  // Return whether the argument is a String
  public function isStringType<T>(t : T) : boolean {
//  public reified function isStringType<T>(t : T) : boolean { // This declaration is also valid
    return t typeis String
  }
}
```

In the method's Gosu code, the symbol `T` can be used as a type and Gosu matches `T` to the type of the argument passed into the method.

The following code uses this class and the parameterized method:

```
var t = new ReificationTester()
print("'abc' is a String: " + t.isStringType("abc"))
print("123 is a String: " + t.isStringType(123))
```

Running this code prints the following messages:

```
'abc' is a String: true
123 is a String: false
```

Generic methods that reify a type

To support parameterized methods that reify the generic type, Gosu provides the keyword `reified`. This keyword instructs Gosu to make information about the runtime type of the type parameter available. Gosu requires the keyword `reified` as a modifier on generic functions that are equivalent to parameterized Java functions that have

type erasure. Use the modifier `reified` on a parameterized function that has any of the following characteristics, where `T` is the token for the type parameter of a generic type:

- The method uses the `new` keyword to create an instance of `T`.
- The method uses the `new` keyword to create an instance of a generic type and includes the `<T>` qualifier on the type name.
- An expression in the method uses the generic type with the `typeis` or `typeof` operator.
- The method contains code that uses `as T` to cast a value to the generic type.
- The method overrides a reified method in a parent class or implements a reified method in an interface.
- The method calls another reified method and passes an argument of a generic type to that method.

The modifier `reified` is optional on a parameterized method that does not have any of these characteristics.

Gosu forbids the modifier `reified` on methods that are not generic.

For a function that requires this modifier, the signature looks like the following examples:

```
reified function cast<N>(type : Type<N>) : List<N>
reified function partition<Q>( partitioner(elt : T):Q ) : Map<Q, List<T>>
static reified function returnLast<T>(a : ArrayList<T>) : T
```

Although some parameterized functions do not require the `reified` keyword, good practice is to make such methods reified. Using the `reified` keyword prevents error conditions if future changes to the code in the method or code that overrides the method cause reification of the parameterized type.

Using a generic collection in a generic method

A generic method that uses `new` to create a generic collection requires the `reified` keyword only if the method instantiates an object of the generic type. The `reified` keyword is optional if the new collection is empty. In this case, Gosu does not require the type of the collection. The `reified` keyword is also optional if the method populates the collection with existing objects. In this case, Gosu infers the type of the collection from the types of the objects. For example:

```
// reified is optional because the ArrayList is empty
reified function emptyListE<E>() : List<E> {
  return new ArrayList<E>()
}

// reified is optional because the code populates the ArrayList with an existing object
function oneListFromE<E>(e : E) : List<E> {
  return new ArrayList<E>({e})
}

// reified is required because the code creates a new object of type E to populate the ArrayList
reified function oneListE<E>() : List<E> {
  return new ArrayList<E>({new E()})
}
```

Example: Basic use of a reified method

In the following code, the class is not parameterized but one method is parameterized. Casting `this` to `T` requires the method to be reified:

```
class BasicReificationTester {
  public reified function returnAsT<T>(a : ArrayList<T>) : T {
    return this as T
  }
}
```

In the method's Gosu code, the symbol `T` is used as a type. Gosu matches `T` to the type of the collection passed into the method.

The following code uses this class and the reified method:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "123"}
var t = new BasicReificationTester()
```

```
var stringT = t.returnAsT(myStrings)

print("Returned item type: " + typeof stringT)
```

The variable `stringT` is strongly typed as `String`, not `Object`.

Example: Reification by accessing type properties

Accessing Gosu properties of the parameterized type requires reification. The following method accesses a Gosu property of the *T* type and requires the `reified` keyword:

```
reified function typeofClassT<T>(t : T) {
  var typ = T.TypeInfo
}
```

Example: Parameterized method that does not require the reified keyword

The following method is parameterized but does not require reification, so the `reified` keyword is optional:

```
// Return whether the argument is a String
public function isStringType<T>(t : T) : boolean {
  return t typeis String
}
```

If future code changes to the code in this method or code that overrides this method cause reification to be necessary, the methods would not compile. Good practice is to use the `reified` keyword.

```
// Return whether the argument is a String
public reified function isStringType<T>(t : T) : boolean {
  return t typeis String
}
```

Example: Another parameterized method that does not require the reified keyword

In the following example, the class `Genericstest` is parameterized:

```
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]
  }
}
```

The following method is parameterized but does not require reification, so the `reified` keyword is optional:

```
// Create an unspecified type of GenericsTest
public function makeANewGenericOfUnspecifiedType<T>(aList : ArrayList<T>) {
  var x = new GenericsTest().printAndReturnFirst(aList)
}
```

Because the code does not specify the type of `GenericsTest`, Gosu uses the the lowest possible bound of `GenericsTest` for the type of the variable *x*, which is `Object`.

If future modifications to the code in this method or code that overrides this method cause reification to be necessary, the methods would not compile. Good practice is to use the `reified` keyword.

Restricting generics

You can use restrictions on generics to define a parameter that supports a specific set of types. For example, you might want to support homogeneous collections or only instances of a class and its subclasses or subinterfaces. A *homogeneous collection* is one in which all items are of the same type.

Consider a custom class `Shape`. You might need a method to support collections of circle shapes or collections of line shapes, where both `Circle` and `Line` classes extend the `Shape` class. For this example, assume the collections are always homogeneous and never contain a mix of both types. You might attempt to define a method like this:

```
public function DrawMe(shapeArray : ArrayList<Shape>)
```

The method accepts an argument of only type `ArrayList<Shape>`. Your code would not compile if you tried to pass an `ArrayList<Circle>` to the method, even though `Circle` is a subclass of `Shape`.

You can specify support of multiple types but limit support to certain types and types that extend those types. Use the syntax "`extends` *TYPE*" after the wildcard character. In Java, this restriction is known as *bounded generics*.

For example:

```
public function DrawMe(shapeArray : ArrayList<T extends Shape>)
```

A valid argument for the parameter `shapeArray` is an `ArrayList` containing objects that are all of the same type, and that type extends the `Shape` class.

Example: Generic class using a restricted bound for its type parameter

The following class restricts the generic type to types that extend `Number`:

```
class BoundedGenericsTest<T extends Number> {
  // 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]
  }
}
```

The following sections use this class in code examples.

Instantiating a generic class that is restricted

You can instantiate a generic class that is restricted in the following ways:

- Using the type token for the type parameter requires information about the type at run time. If you instantiate a generic class in this way in a parameterized method, you must use the `reified` keyword in the method declaration. This instantiation creates a strongly typed object that uses the explicit type that is reified by the method. For example, in the following lines of code, the type of the variable *x* is `BoundedGenericsTest<T>` where `T` is the type of the method parameter `t`:

  ```
  public reified function makeANewGenericOfSpecifiedType<T extends Integer>(t : T) {
    var x = new BoundedGenericsTest<T>()
  }
  ```

- Specifying the type of the type parameter creates a strongly typed object that uses the explicit type. For example, in the following line of code, the type of the variable *x* is `BoundedGenericsTest<Integer>`:

  ```
  var x = new BoundedGenericsTest<Integer>()
  ```

- Omitting the type parameter creates a strongly typed object that uses the least restrictive type of the bound. For example, in the following line of code, the type of the variable *x* is `BoundedGenericsTest<Number>`:

  ```
  var x = new BoundedGenericsTest()
  ```

  You can include this instantiation of `BoundedGenericsTest` in a parameterized method without using the `reified` keyword because Gosu substitutes the least restrictive type of the specified bound for the type parameter.

Restricting parameterized enhancements of classes

If you create an enhancement of a parameterized class that applies to a restricted set of types, you must use the correct syntax. For example, consider a parameterized class, `GenericsTest` that has the following declaration:

```
class GenericsTest<T>
```

For an enhancement that applies to a restricted set of type on the generic class, you specify the restrictions on the enhancement name and the type wildcard on the class name. For example, the declaration of an enhancement to types that extend `CharSequence` types of the `GenericsTest` class looks like the following line:

```
enhancement GenericsTest_CharSeqEnh<T extends CharSequence> : GenericsTest<T>
```

See also

- "Enhancements" on page 201

Using generics with collections and blocks

Common uses of generics apply them to collections and to block functions. You can combine these uses of generics to create API methods that are relevant for many purposes. Blocks are anonymous functions that can be passed as parameter objects to other methods.

If a method takes a block as a parameter, you can use generics to describe the set of blocks that the method supports. The parameter definition specifies the number of arguments, the argument types, and the return type of the block. For example, consider a block that takes a `String` parameter and returns another `String`. The definition of the block itself indicates one argument, the parameter type `String`, and the return type `String`. In the definition of the block parameter on the method, you can use generics to specify how the parameter to the block and its return type match the type on the parameterized class.

For example, Gosu provides an enhancement to the Java `List` class that provides additional functionality to any collection. This enhancement provides multiple methods that rely on generics to determine the type of the collection. A typical example of this use of generics is the list enhancement method `sortBy`, which takes a block parameter. That block takes exactly one argument, which must be the same type as the items in the list. For example, if the list type is `ArrayList<String>`, the block must take a `String` parameter. The following code shows the method definition in the enhancement.

```
enhancement CoreListEnhancement<T> : List<T> {
...
  function sortBy( value(elt : T):Comparable, comparator : Comparator =  null) : List<T>
  {
    ...
  }
}
```

In the definition of the enhancement, the method, and the block parameter, the symbol `T` is treated as the type of the collection. Note the use of the letter `T` in the block signature:

```
value(elt : T):Comparable
```

That signature specifies that the block parameter takes one argument of type `T` and returns a `Comparable` value, such as an `Integer` or `String`.

Consider the following array list of strings:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}
```

You can re-sort the list based on the length of the strings by using a block. Create a block that takes a `String` parameter and returns the sort key, in this case the text length. The `List.sortBy(...)` method implements the sorting algorithm to sort the original list in place, and does not return an entirely new list.

The following example uses the `sortBy` method to sort a list of `String` objects by the length of the `String`, and then prints the sorted list values.

```
var resortedStrings = myStrings.sortBy( \ str -> str.length() )
resortedStrings.each( \ str -> print( str ) )
```

This code prints:

```
a
ab
abc
abcd
```

Using generics for type inference in cases like this example produces readable and concise Gosu code.

This example omitted the types of the block argument `str` and the sort key. You do not have to specify these types explicitly. Although the following code is valid, Gosu uses type inference to determine the types.

```
var resortedStrings = myStrings.sortBy( \ str : String -> str.length() as Integer )
```

See also

- "Blocks" on page 175
- "Collections" on page 183

Multiple dimensionality generics

Typical use of generics is with one-dimensional objects, such as lists of a certain type of object. For example, a list of `String` objects or a collection of `Address` objects are one-dimensional.

Gosu generics are flexible and support multiple dimensions, just like Java generics. Generics that specify multiple dimensions can define the use of a `Map`, which stores a set of key-value pairs and is like a two-dimensional collection.

By using generics, you can define behavior for multiple types of maps. For example, you can use two-dimensional generics to define a method signature:

```
public function getHighestValue( theMap : Map<K,V>) : V
```

The parameter `theMap` has type `Map`. Gosu generics uses two type wildcards as single capital letters separated by commas to parameterize the types in the map. By convention, the first wildcard (`K`) represents the type of the key and the second wildcard (`V`) represents the type of the value. Because the method signature uses the `V` again in the return value type, the Gosu compiler makes assumptions about relationships between the type of map and the return value. The two uses of `V` in the method signature specify the same type for both the map value in the parameter and the return value.

At run time, the symbols `K` and `V` are assigned to specific types. The code that calls the method creates new instances of the parameterized class with specific types. The compiler can determine which method to call and the types that `K` and `V` represent.

A specific type of map that has keys of type `Long` and values of type `String` has a definition like the following line of code.

```
Map<Long, String> contacts = new HashMap<Long, String>()
```

If you pass the variable `contacts` to this `getHighestValue` method, the compiler knows that this call to the method returns a `String` value.

You can also define a class that uses multiple dimensions of types. For example, to work with key-value maps, you can use generics to define a class:

```
class Mymapping<K,V> {
  public function put( key : K, value : V) {...}
  public function get( key : K) : V {...}
}
```

In the method definitions, the values in angle brackets have the same meanings as the type names in the parameterized class definition. In this example, the `K` and `V` symbols specify the types. Use these symbols in method signatures in the class to represent types in arguments, return values, and Gosu code inside the method. You can use this class to provide strongly typed results.

In the following example, the concrete types are `String` for `K` and `Integer` for `V`.

```
var myMap = new Mymapping<String, Integer>()
myMap.put("ABC", 29)

var theValue = myMap.get("ABC")
```

The `theValue` variable is strongly typed at compile time as `Integer`.

Because the code is strongly typed at compile time, the reliability of the code improves at run time.

Generics with custom containers

Although Gosu generics are often used with collections and lists, there is no requirement to use these features with built-in `Collection` and `List` interfaces. Any class that represents a container for other objects can use Gosu generics to define the type of items in the container.

Abstract example

Suppose you want to need to store key-value maps. You can define a class that uses the `Object` class to define two types of object:

```
class Mymapping {
  function put( key : Object, value : Object) {...}
  function get( key : Object) : Object {...}
}
```

Alternatively, you can use generics to define the class:

```
class Mymapping<K,V> {
  function put( key : K, value : V) {...}
  function get( key : K) : V {...}
}
```

This class enforces strongly typed values at compile time. In the following code, the `theValue` variable is strongly typed at compile time as `Integer`.

```
var myMap = new Mymapping<String, Integer>
myMap.put("ABC", 29)

var theValue = myMap.get("ABC")
```

Real-world example

Consider a program that tracks the construction of vehicles within multiple factories. You can represent cars with `Car` objects, trucks with `Truck` objects, and vans with `Van` objects. All these classes derive from a root class `Vehicle`.

Your program uses custom container object of type `FactoryGroup` that does not derive from the built-in collection classes. For this example, assume that each factory only contains one type of vehicle. A `FactoryGroup` can contain multiple `Car` objects, or multiple `Truck` objects, or multiple `Van` objects.

You need an API 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 can represent these groups by using the syntax:

- `FactoryGroup<Car>`
- `FactoryGroup<Truck>`
- `FactoryGroup<Van>`

Consider a method that returns all vehicles in the last step in a multistep manufacturing process. The method definition is like the following line:

```
public function getLastStepVehicles(groupofvehicles : FactoryGroup<T>) : FactoryGroup<T>
```

By using generics, the method supports all types of `FactoryGroup` objects. Because the letter `T` appears more than once in the method signature, this syntax defines relationships between the parameter and the return value. The method `getLastStepVehicles` takes one argument that is a factory group containing any one vehicle type. The method returns another factory group that is guaranteed to contain the same type of vehicle.

Alternatively, you could define your API with bounded wild cards for the type:

```
public function getLastStepVehicles(groupofvehicles : FactoryGroup<T extends Vehicle>) : FactoryGroup<T>
```

By using this approach, your code can make more assumptions about the type of object in the factory group. This approach also prevents some coding errors, such as accidentally passing `FactoryGroup<String>` or `FactoryGroup<Integer>`, which fail at compile time. You can discover your coding errors quickly.

To use this approach, you must use Gosu syntax to inform the compiler that your class is a container class that supports generics. Add the bracket notation in the definition of the class, and use a capital letter to represent the type of the class. For example, a standard class definition is:

```
public class FactoryGroup
```

The definition of a class as a container class supporting generics uses the syntax:

```
public class FactoryGroup<T>
```

chapter 17

Dynamic types and expando objects

Gosu supports dynamic language features that permit coding styles similar to non-statically typed languages. The `dynamic.Dynamic` type allows dynamic assignment as well as dynamic dispatch of property access and method invocation. Gosu also provides support for *expando objects*, which simplify dynamic property and method access for typical code contexts.

Overview of dynamic language features

Gosu is a statically typed language employing a nominal type system. A *nominal type system* is one in which type assignability is based on declared type names, type ancestry, and declared interfaces. Because Gosu is statically typed, in general programming, access to properties is dependent on the property definitions of the declared class. In typical code, these restrictions encourage coding styles that catch most types of programming errors at compile time, and support refactoring code safely.

In rare cases, dynamic coding styles can be useful. The Gosu type system supports two features that expand support for dynamic type access:

- **Dynamic type declaration** – Declare a variable or function parameter as the type `dynamic.Dynamic` to allow assigning any reference type. This type declaration enables dynamic dispatch of property and method invocation by the name of the property or method.
- **Expando object support** – Expando objects are objects that use the dynamic declaration and store properties as key-value mappings in a `javax.script.Bindings` object. That class extends the `java.util.Map` class.

You can use these features to use coding styles that are common in languages that are not statically typed.

WARNING Use dynamic language features judiciously because misuse can hide errors that the Gosu editor cannot catch at compile type. For example, if you misspell a property or method name, there is no compiler error but run time errors can occur.

Note that these two features are separate from structural types. *Structural types* are similar to interfaces but can be used with objects with no shared ancestry or interfaces.

See also

- "Structural types" on page 241

Dynamic type

In a variable declaration or function parameter, you can declare the type as the type `Dynamic`. The `Dynamic` type permits any type of object to be assigned to that variable. The fully qualified name is `dynamic.Dynamic`. At first

glance, this might be similar to using the `Object` type, which is the base class of all reference types. However, there are important differences.

By declaring the `Dynamic` type for the variable or function argument, you can programmatically access any property or method without requiring compile-time confirmation of its existence. For example, get the `Name` or `Age` property from an object even if it does not exist at compile time. In contrast, if you use the `Object` type, the Gosu compiler validates any property or method on the `Object` class itself, which does not have many properties or methods.

On its own, the dynamic feature can be used for dynamic programming styles where property names and method names may not be known at compile type. To dynamically get properties, set properties, and invoke methods, you can optionally implement special methods on the target class directly on the declaring class or with a Gosu enhancement.

WARNING Use dynamic language features judiciously because misuse can hide errors that the Gosu editor cannot catch at compile type. For example, if you misspell a property or method name, there is no compiler error but run time errors can occur.

The following table lists the special methods you may implement on classes who objects are accessed by variables declared with the `dynamic.Dynamic` type. In the following table, the term `UNHANDLED` refers to value `gw.lang.reflect.IPlaceholder.UNHANDLED`.

Dynamic type method name	Description
`$getProperty`	Gets a property by its name specified as an argument. If the property is unknown, return the value `UNHANDLED`. If you return `UNHANDLED`, Gosu calls the `$getMissingProperty` method, which must exist or Gosu throws an exception.
`$getMissingProperty`	Similar to `$getProperty`, but Gosu only calls this method if the type does not explicitly declare the property. Gosu calls this method if some code sets a property and either of the following is true: • The `$getProperty` method is absent and the property is not defined explicitly on the class • The `$getProperty` method exists and it returned `UNHANDLED`.
`$setProperty`	Sets a property by its name specified as an argument, with the value in the second argument. If the property is unknown, return the value `UNHANDLED`. If you return `UNHANDLED`, Gosu calls the `$setMissingProperty` method, which must exist or Gosu throws an exception.
`$setMissingProperty`	Similar to `$setProperty`, but Gosu only calls this method if the type does not explicitly declare the property. Gosu calls this method if some code sets a property and either of the following is true: • The `$setProperty` method is absent and the property is not defined explicitly on the class • The `$setProperty` method exists and it returned `UNHANDLED`.
`$invokeMethod`	Invokes a method by its name specified as an argument, with the parameter list in the second argument as an array of `Object` instances. If the method is unknown or the arguments have the wrong number or types, return the value `UNHANDLED`. If you return `UNHANDLED`, Gosu calls the `$invokeMissingMethod` method, which must exist or Gosu throws an exception.
`$invokeMissingMethod`	Similar to `$invokeMethod`, but Gosu only calls this method if the type does not explicitly declare the method. Gosu calls this method if some code invokes a method and either of the following is true: • The `$invokeMethod` method is absent and the method with matching argument number and types is not defined explicitly on the class • The `$invokeMethod` method exists and it returned `UNHANDLED`.

The following example defines a simple object with a dynamic property getter method. In the example, the `$getProperty` method handles two property names explicitly, and then returns `UNHANDLED` for any other property names. Note that for the unhandled property name, Gosu automatically calls the `$getMissingProperty` method.

In real-world code, the `$getProperty` method might get the value from a `java.util.Map`, from a related subobject, from a calculation, or even from an external system:

```
package doc.example

uses gw.lang.reflect.IPlaceholder
uses dynamic.Dynamic

class DynamicGetter {

  // Handle any property name. Dynamically return
  // values for two values, and return UNHANDLED for others
  public function $getProperty(fieldName: String): Object {
    print("$getProperty called with field: " + fieldName)

    if (fieldName == "StreetAddress") {
      return "123 Main Street"
    } else if (fieldName == "FirstName") {
      return "John"
    } else {
      return IPlaceholder.UNHANDLED
    }
  }

  // Handle any unhandled requests for missing property names
  public function $getMissingProperty(fieldName: String): Object {
    print("$getMissingProperty called with field: " + fieldName)
    return ("fakeValue")
  }
}
```

The following code uses the dynamic object and gets a property from it:

```
uses doc.example.DynamicGetter
uses dynamic.Dynamic

var obj : Dynamic = new DynamicGetter()

print("---")
print("Dynamically get a known property...")
print("RESULT = " + obj.StreetAddress)

print("---")
print("Dynamically get a missing property...")
print("RESULT = " + obj.OtherField)
```

This code prints the following:

```
---
Dynamically get a known property...
$getProperty called with field: StreetAddress
RESULT = 123 Main Street
---
Dynamically get a missing property...
$getProperty called with field: OtherField
$getMissingProperty called with field: OtherField
RESULT = fakeValue
```

The dynamic type works well with the `IExpando` interface and `Expando` objects. Use these objects together to create dynamic property getters and setters and store data in a `java.util.Map` object.

See also

- “Expando objects” on page 235
- “Structural types” on page 241
- “Enhancements” on page 201

Expando objects

An expando object is an object whose properties, and potentially methods, are created dynamically on assignment. Most dynamic languages provide what are called *expando objects*. For object oriented programming, some

languages provide only expando objects. An expando object behaves a lot like a map, in that you associate named keys with values. In contrast to a map, with expando objects the keys appear as natural properties directly on the object.

IExpando interface

To enable dynamic programming with a map-based implementation, Gosu provides the `IExpando` interface, which contains the following method and property declarations. Any class that implements this interface can behave as an expando object in the context of a variable declared as the `Dynamic` type.

The base configuration provides the `Expando` class, which is an implementation of this interface.

Get field value

```
override function getFieldValue(s: String): Object
```

The `getFieldValue` method gets a field and returns an `Object`.

Set field value

```
override function setFieldValue(s: String, o: Object)
```

The `setFieldValue` method sets a field with an `Object`.

Invoke method

```
override function invoke(s: String, objects: Object[]): Object
```

The `invoke` method invokes a method with zero, one, or more arguments passed as an array.

Set default field value

```
override function setDefaultFieldValue(s: String)
```

The `setDefaultFieldValue` method sets a field default value. Gosu calls this method in certain cases where Gosu code tries to get an uninitialized field value. The only situation where Gosu calls the method is during automatic creation of intermediary objects in an object path. For example:

```
var dyn = new Dynamic()
dyn.abc.def = "hi"
print( dyn.abc.def ) // prints "hi"
```

In this example, note the middle line. Because the `dyn.abc` property does not exist yet, and it is necessary to set the `abc.def` property, Gosu creates the intermediate object. During this automatic creation process, Gosu calls the `setDefaultFieldValue` to create the default value for `abc`. In the built-in `Expando` implementation, this method always creates another instance of the `Expando` class.

See also

- “Expando objects” on page 235
- “The Expando class” on page 236
- “Property assignment triggering instantiation of intermediate objects” on page 56

The Expando class

Gosu provides a default implementation of the `IExpando` interface called `gw.lang.reflect.Expando`. This default implementation delegates to a `java.util.HashMap` for data storage.

The property name is always a `String` value, the value is an `Object`. If you get an uninitialized property, the value is `null`.

For any methods, the value must be a Gosu block. If you try to call a method that does not exist, the method returns the special value `IPlaceholder.UNHANDLED`.

The following example creates an `Expando` instance in a variable declared to type `Dynamic`.

```
uses gw.lang.reflect.Expando
uses dynamic.Dynamic

// Create new Expando object in a Dynamic variable
var villain : Dynamic = new Expando()

// Set dynamic properties and methods, which are stored internally in a java.util.HashMap
villain.Health = 10
villain.punch = \-> { if ( villain.Health > 0 ) villain.Health-- }
villain.isDead = \-> villain.Health <= 0

// Use the Expando object with a natural, readable syntax:
print("Health = " + villain.Health)

while ( !villain.isDead() ) {
  villain.punch()
  print("After punch, health = " + villain.Health)
}

print("Health = " + villain.Health)
```

This example prints:

```
Health = 10
After punch, health = 9
After punch, health = 8
After punch, health = 7
After punch, health = 6
After punch, health = 5
After punch, health = 4
After punch, health = 3
After punch, health = 2
After punch, health = 1
After punch, health = 0
Health = 0
```

See also

- “Blocks” on page 175
- “Dynamic type” on page 233
- “Structural types” on page 241
- “Enhancements” on page 201

Concise syntax for dynamic object creation

Gosu supports a concise object creation syntax similar to JSON directly in its grammar. The following Gosu features provide this functionality:

- You can omit the type name in a `new` expression if Gosu can infer the type. For example, instead of typing `new TestClass(arg1, arg2)`, you can type `new(arg1, arg2)`. The programming context already assumes the compile-time type called `TestClass`.
- You can initialize fields during instantiation using a special syntax with curly braces around a comma-delimited list of property assignments, with a colon before the property name. For example:

  ```
  new TestClass() { :Name = "John Smith" }
  ```

- Gosu supports dynamic non-type-safe access to properties and methods for variables of the type called `dynamic.Dynamic`. No declaration of specific property names or method names is required in advance. For

programming contexts of type `dynamic.Dynamic`, a newly instantiated object assumes the default run-time type of `Expando`. The `Expando` class implements the `Bindings` interface. Gosu adds enhancement methods to this class to convert to JSON (`toJson`), XML (`toXml`), or Gosu object initialization code (`toGosu`).

When you combine these features, you can create objects with multiple types of data without declaring any specific structure in advance, and can convert to other formats as needed.

The following Gosu code creates a dynamic complex object and prints it in JSON syntax, XML syntax, and Gosu object initialization syntax:

```
var person: dynamic.Dynamic

person = new() {
  :Name = "John Smith",
  :Age = 39,
  :Address = new() {
    :Number = 123,
    :Street = "Main St.",
    :City = "Foster City",
    :State = "CA"
  },
  :Hobby = {
    new() {
      :Category = "Sport",
      :Name = "Swimming"
    },
    new() {
      :Category = "Recreation",
      :Name = "Hiking"
    }
  }
}

print("**** To JSON:")
print(person.toJson())

print("**** To XML:")
print(person.toXml())

print("**** To Gosu:")
print(person.toGosu())
```

This example prints the following:

```
**** To JSON:
{
  "Name": "John Smith",
  "Age": 39,
  "Address": {
    "Number": 123,
    "Street": "Main St.",
    "City": "Foster City",
    "State": "CA"
  },
  "Hobby": [
    {
      "Category": "Sport",
      "Name": "Swimming"
    },
    {
      "Category": "Recreation",
      "Name": "Hiking"
    }
  ]
}
**** To XML:
<object>
  <Name>John Smith</Name>
  <Age>39</Age>
  <Address>
    <Number>123</Number>
    <Street>Main St.</Street>
    <City>Foster City</City>
    <State>CA</State>
  </Address>
```

```
  <Hobby>
    <li>
      <Category>Sport</Category>
      <Name>Swimming</Name>
    </li>
    <li>
      <Category>Recreation</Category>
      <Name>Hiking</Name>
    </li>
  </Hobby>
</object>
**** To Gosu:
new Dynamic() {
  :Name = "John Smith",
  :Age = 39,
  :Address = new() {
    :Number = 123,
    :Street = "Main St.",
    :City = "Foster City",
    :State = "CA"
  },
  :Hobby = {
    new() {
      :Category = "Sport",
      :Name = "Swimming"
    },
    new() {
      :Category = "Recreation",
      :Name = "Hiking"
    }
  }
}
```

Note that for the Gosu initializer syntax serialization, you can evaluate the `String` data at run time to create a Gosu object:

```
var clone = eval( person.toGosu() )
```

See also

- “Optional omission of type name with the new keyword” on page 87
- “Object initializer syntax” on page 88
- “Dynamic types and expando objects” on page 233

chapter 18

Structural types

Use structural typing to write code that works with objects with similar features but no common inheritance and interface declarations. Define structural types similar to defining interfaces, by specifying the common properties and method signatures. However, use the `structure` keyword, not the `interface` keyword.

Structural types are weaker than interfaces with respect to the amount of enforced type information they have in them. However, their flexibility supports situations where interfaces are ineffective or impossible. Structural types extend static typing to include a broader set of real-world situations but still support concise code that catches common coding problems at compile time.

Overview of structural types

Gosu is a statically typed language employing a nominal type system. A *nominal type system* is one in which type assignability is based on declared type names, type ancestry, and declared interfaces. For instance, in Gosu a type called `Test1` is assignable to interface `Interface1` only if `Test1` declares `Interface1` in its hierarchy explicitly.

Another form of static typing called *structural typing* determines assignability based on declared type features, such as methods and properties. The type called `B` is structurally assignable to a structural type `StructA` if B has compatible versions of all the methods and properties of `StructB`. The type `B` does not have to formally declare that it implements `StructA`. Structural typing is more flexible than nominal typing because it measures a type based on its capability, not its name or inheritance declarations.

Limitations of nominal typing

Strictness with nominal typing is not always desirable. Consider the following three nominally unrelated types that might be part of a third-party Java library whose code you do not control:

```
// Three difference classes that have X and Y properties
// but they have no shared inheritance nor shared interfaces that define X and Y

class UIObject {
  public var X : double
  public var Y : double
  public var H : double
  public var W : double
}
class Rectangle {
  private var _x : double as X
  private var _y : double as Y
  private var _w : double as Width
  private var _h : double as Height
}
class Point {
  private var _x : double as X
```

```
  private var _y : double as Y
}
```

Suppose you want to write code that just gets the `X` and `Y` properties and you want your code to work with instances of any of the three classes.

Although each of these classes defines methods for obtaining the `X` and `Y` properties, these properties do not appear as part of a common interface or ancestor class. Remember that in this example, the classes are defined in a third-party library, so you cannot control the class declarations. Their only base class in common is `Object`, but `Object` does not have the properties `X` or `Y`. To write code that reads the `X` and `Y` properties for these three different classes, you must write three distinct yet nearly identical versions of the code.

Situations like this are common. Typically programmers resolve the problem by duplicating potentially large amounts of code for each related use cases, which makes it more difficult to maintain the code.

Structural typing helps these types of situations. Use structural typing in Gosu to unite the behavior of the three classes based on the presence of being able to read the `X` and `Y` properties.

Just like you would declare an interface, define the method signatures. However, use the `structure` keyword:

```
package docs.example

structure Coordinate {
  property get X() : double // get the "X" public property or field
  property get Y() : double // get the "Y" public property or field
}
```

You can now create code that uses the new structural type `docs.example.Coordinate`.

A variable declared as the structural type can contain any object with compatible qualities defined by the structural type. For example, a variable defined to the structural type `Coordinate` can be assigned with a `Rectangle` instance. This works because `Rectangle` has the `X` and `Y` properties of type `double`:

```
var coord : Coordinate
coord = new Rectangle()
print("X value is " + coord.X)
```

To illustrate the flexibility of structural typing, suppose you want to write a function that gets the `X` and `Y` properties and prints them and performs arithmetic with the values. Using structural typing, implementing this behavior once works with any class instance that satisfies the requirements of the structural type. For example, the following function takes the structural type as a parameter:

```
public function printCoordinate(myStruct : Coordinate) {
  print("X value is " + myStruct.X)
  print("Y value is " + myStruct.Y)
  print("Sum of X and Y value is " + (myStruct.X + myStruct.Y) )
}
```

In many ways this coding style is similar to using an interface defined by the `interface` keyword. You can call this function and pass any object that satisfies the requirements of the structural type. However, unlike interfaces, this code works independent of inheritance or interface declarations for the classes.

Use the following example to see how this function can be used by multiple concrete classes that are unrelated. Paste the following into the Gosu Scratchpad:

```
// Three different classes that have X and Y properties
// but have no shared inheritance nor shared interfaces that define X and Y

class Rectangle {
  private var _x : double as X
  private var _y : double as Y
  private var _w : double as Width
  private var _h : double as Height
}

class Point {
  private var _x : double as X
  private var _y : double as Y
```

```
}

class UIObject {
  public var X : double
  public var Y : double
  public var H : double
  public var W : double
}

// DEFINE A STRUCTURAL TYPE
structure Coordinate {
  property get X() : double
  property get Y() : double
}

// Define a function that works with any class that is compatible with the structural type
public function printCoordinate(myStruct : Coordinate) {
  print("X value is " + myStruct.X)
  print("Y value is " + myStruct.Y)
  print("Sum of X and Y value is " + (myStruct.X + myStruct.Y) )
}

// Use that function with three different classes that all contain X and Y public fields
// or properties, but are unrelated

var p = new Point()
p.X = 2
p.Y = 3
printCoordinate(p)

var r = new Rectangle()
r.X = 4
r.Y = 5
printCoordinate(r)

var ui = new UIObject()
ui.X = 6
ui.Y = 7
printCoordinate(ui)
```

Running this code prints the following lines:

```
X value is 2.0
Y value is 3.0
Sum of X and Y value is 5.0
X value is 4.0
Y value is 5.0
Sum of X and Y value is 9.0
X value is 6.0
Y value is 7.0
Sum of X and Y value is 13.0
```

These examples demonstrate that structural types are statically weaker than interfaces regarding the amount of enforced type information, but their flexibility supports situations where interfaces are ineffective or impossible. Structural types extend static typing to include a broader set of real-world situations but still support concise code that catches common coding problems at compile time.

Create a structural type with getters, setters, and methods

About this task

To create a structural type, you perform the following general steps:

Procedure

1. In the package for your new type, create a Gosu class file with a `.gs` file extension.
2. In the Gosu editor, change the word `class` to `structure`.

3. Just as you would declare on an interface, add property declarations and method signatures:
   - To provide data from a variable or property on an instance of this type, use property getter syntax:

     ```
     property get PROPERTY_NAME() : TYPE
     ```

   - To set the value of a variable or property on an instance of this type, use property setter syntax:

     ```
     property set PROPERTY_NAME(TYPE)
     ```

   - To provide a callable method on an instance of this type, use the syntax:

     ```
     public function METHOD_NAME(ARG_LSIT) : TYPE
     ```

Example

```
package docs.example

structure DemoStructure{
  property get Name() : String
  property set Name(n : String)
  public function count(s : String) : double
}
```

This example declares a structural type called `DemoStructure` that has the following features:

- `Name` property of type `String`
- `count` method, which takes one `String` parameter and returns a `double` value

See also

- “Interfaces” on page 171
- “Adding static constant variables to a structural type” on page 244

Adding static constant variables to a structural type

In both Gosu and Java, you can add static variables to types to declare data that exists at the type level. For Gosu classes, you add such data to a class by adding a declaration line with the `static` modifier:

```
public static var MyVariable
```

For structural type definitions, any variables that store state are implicitly static and constant so you can omit the word `static`:

```
public var MyVariable
```

Adding a `var` declaration does not imply the ability to get or set a variable or property on an instance of the type. A `var` declaration on a structural type defines a static constant variable on the type. A static constant variable is conceptually similar to a static variable on a Gosu class. However, as for a variable on an interface, the `static` keyword is implicit and the value is read-only. The declaration does not specify the ability to get or set a variable or property an instance of this type.

For example, note the `Age` variable in the following example:

```
package docs.example

structure DemoStructure2 {
  property get Name() : String
  property set Name(n : String)
  public var Age : Integer      // Compilation error. A static constant variable requires initialization
}
```

The `var` declaration on the structural type creates a static constant variable, which must be initialized, so the previous code causes a compilation error. Initializing the value of the static constant variable clears the error:

```
package docs.example

structure DemoStructure2 {
  property get Name() : String
  property set Name(n : String)
  public var AGE : Integer = 18 // OK because the static constant variable is intialized
}
```

You can add the `static` modifier in the declaration of the `AGE` variable, but in structural type declarations, this modifier is implicit and therefore redundant.

Although you can use property getter and setter syntax to define data to get and set on instances, you do not use this syntax for static constant variables. Therefore, static constant variables do not require assignment compatibility between the structural type and other types.

The `DemoStructure2` class from the previous example is compatible with the following class:

```
package docs.examples

class MyClass {
  public var Name: String
  public var UnusedField1: String
  public var UnusedField2: String
}
```

Notice the following characteristics of `MyClass`:

- The Gosu class `MyClass` has a `Name` variable, which is the only requirement to be compatible for assignment to a variable of the structural type `DemoStructure2`. Because `MyClass` is a Gosu class and not a structural type, the `Name` variable is a natural instance variable and is not static. Instance variables and properties on a concrete type are compatible with getters and setters on the structural type, so these are compatible names and types.
- The Gosu class `MyClass` does not have an `AGE` variable or property. Because the `AGE` variable on the structural type is a static constant, it exists on the structural type only. For a static constant variable, the concrete type does not require compatibility for assignment between the types.

For example, you can now run the following code in the Gosu Scratchpad:

```
uses docs.examples.DemoStructure2

uses docs.examples.MyClass

var obj : MyClass
var struct : DemoStructure2

// Create a new instances of the concrete type
obj = new MyClass()

// Assign the MyClass object to a variable declared as the structural type
struct = obj

// Set the shared field on the variable of the structural type, and it affects both
struct.Name = "John"
print("struct.Name is " + struct.Name)
print("obj.Name is " + obj.Name)

// Set the shared field on variable of type MyClass, and it affects both
obj.Name = "Eric"
print("struct.Name is " + struct.Name)
print("obj.Name is " + obj.Name)

// struct.AGE is a STATIC CONSTANT and is not shared, and is read-only
print("struct.AGE (a STATIC CONSTANT) is " + struct.AGE)

// The following line is not valid if you uncomment it. Static constant fields are READ-ONLY.
// struct.AGE = 40
```

This code prints the following:

```
struct.Name is John
obj.Name is John
struct.Name is Eric
obj.Name is Eric
struct.AGE (a STATIC CONSTANT) is 18
```

See also

- “Create a structural type with getters, setters, and methods” on page 243

Assignability of an object to a structural type

If a variable is declared as a structural type, any object whose type satisfies the structural type can be assigned to that variable. The participating members of a structural type are:

- Property getters
- Property setters
- Methods

Adding a `var` declaration to a structural type does not specify the ability to get or set a variable or property an instance of this type. Instead, a `var` declaration defines a static constant variable on the type. To get or set data from a variable and properties on an instance of the structural type, you must use the property getter and setter syntax in its declaration.

Property assignment compatibility

For properties, assignment compatibility extends to primitive types. For example, if a structure defines a property as type `double`, a compatible property can define it as a class that defines it as an `int`. Because an `int` can coerce to a `double` with no loss of precision, the property is call-compatible.

Method assignment compatibility

For methods, given type `T` and structure `S`, a `T` method is structurally assignable to an identically named method in `S` if all of the following are true:

- The number of parameters in both methods is the same.
- The parameter types of the method on `T` are assignable from the parameter types of the method on `S`.
- The return type of the method on `T` is assignable to the return type of the method on `S`.

In other words, the parameter types are contravariant and the return type is covariant.

Method signature variance on structure methods supports primitive types as well as reference types.

Interface assignment compatibility

If you define a structure that extends an interface, the structure gets the properties and method signatures of the interface. The structural assignability rules stated earlier apply, rather than interface assignability rules.

See also

- “Adding static constant variables to a structural type” on page 244
- “Create a structural type with getters, setters, and methods” on page 243

Assignability of a structural type to another type

A structure is assignable to variables and programming contexts of the following types:

- A structure of the same type
- Other structures with compatible assignable properties and compatible method signatures
- `Object`

Using enhancements to make types compatible with structures

Gosu enhancements are a language feature to add methods and properties to other types even if you do not control the implementation classes for those types. Programming access to enhancements methods and properties are dispatched statically at compile time based on the programming context. For example, if you add the `toPublicName` method to the type `MyClass`, every programming context for an instance of `MyClass` now supports the `toPublicName` method.

You can use Gosu enhancements to add methods and features to types that make the types compatible with structural types. For example, suppose you want a structural type to work with multiple types that have a `toPublicName` method that returns a `String` value. You might create a structure such as:

```
structure HasPublicName {
  public function toPublicName() : String
}
```

Suppose a relevant class does not provide the `toPublicName` method but exposes the appropriate data using other methods or properties. You can add a Gosu enhancement to add the desired property on target types, in this case instances of the `com.example.MyClass` class:

```
enhancement AddDisplayNameToString: com.example.MyClass {
  public function toPublicName() : String {
    return this.toString()
  }
}
```

Any instance of `com.example.MyClass` has the `toPublicName` function and is compatible with the structure that requires that method:

```
uses com.example.MyClass

var s = new MyClass()      // Create a new String

var named : HasPublicName  // Variable declared to a structural type

// Assign a MyClass to the structural type. This code works because of the Gosu enhancement method.
named = s
```

See also

- “Enhancements” on page 201

chapter 19

Composition

Gosu provides the language feature called composition by using the `delegate` keyword in variable definitions. Composition enables a class to delegate responsibility for implementing an interface to a different object. This compositional model supports implementation of objects that are proxies for other objects, or encapsulation of shared code independent of the type inheritance hierarchy.

See also

- "Interfaces" on page 171
- "Classes" on page 147

Using Gosu composition

The language feature *composition* enables a class to delegate responsibility for implementing an interface to a different object. This feature supports code reuse for 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 and can cause changes to a base class to break all subclasses. Languages that support multiple inheritance, which allows 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? Because the classes 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`. No natural place exists 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 behavior 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
```

Optionally, you can use the following syntax that specifies the type:

```
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
```

In the class constructor, create an instance of an object that implements the interface. For example:

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

After the constructor runs, Gosu intercepts any method invocations on the object for that interface and forwards the method invocation to the delegated object.

Example

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 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!") }
}
```

The 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 Scratchpad:

```
uses test.MyWindow

var a = new MyWindow()

// Call a method handled on the delegate
a.paste()
```

This code 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, in the following line, `_clipboardPart` implements the `IClipboardPart` interface for the `ClipboardPart` class:

```
private delegate _clipboardPart : ClipboardPart represents IClipboardPart
```

Using one delegate for multiple interfaces

You can use a delegate to represent multiple interfaces for the enclosing class by implementing the methods for those interfaces. 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`, which represents two different types, which in this case are interface types. You might wonder about the compile-time type of the `_employee` variable. Because the variable must satisfy all requirements of both types, Gosu uses a special type called a compound type. A *literal of a compound 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.

See also

- “Compound types” on page 409

Using composition with built-in interfaces

You can use composition with any interfaces, including built-in interfaces. For example, you could give a custom object all the methods of `java.util.List` and delegate the implementation to an instance of `java.util.ArrayList` or another `List` implementation.

For example:

```
class MyStringList implements List<String> {
  delegate _internalList represents List<String> = new ArrayList<String>()
}
```

You could now use this class and call any method defined on the `List` interface:

```
var x = new MyStringList()
x.add( "TestString" )
```

chapter 20

Working with XML in Gosu

XML files describe complex structured data in a text-based format with strict syntax for 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.

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 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 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.

See also

- "Introduction to the XML element in Gosu" on page 253

Introduction to the XML element in Gosu

The main class that represents an XML element is the class `XmlElement`.

An `XmlElement` object encapsulates the following core:

- The element name as a qualified name, a `QName`
- A backing type instance
- The nillness of the element

Element qualified name

The element's name is a `String` value that 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 within which the local part of the name is defined. 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, because there may be conflicts.

For example, an XML file may contain an element name with the syntax of two parts separated by a colon, such as `veh:childelement`. The `childelement` part of the name is the local part. The prefix `veh` indicates that earlier in the file, the XML document declared a namespace and the prefix `veh` as a shortcut 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>
```

This XML code specifies the following properties of the XML document:

- The root element of the document has the name `root` within the namespace `http://mycompany.com/schema/vehiclexsd`.
- The text `xmlns:veh` text 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, which has the name `childelement` and is within the namespace `http://mycompany.com/schema/vehiclexsd`. However, this XML element specifies the namespace not with the full URI but with the shortcut prefix `veh` followed by the colon, which is 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, with an implicitly empty suggested prefix

```
QName(String namespaceURI, String localPart)
```

- Constructor specifying the local part only, with implicitly empty URI and namespace

```
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 in the same namespace, you can use the optional utility class called `XmlNamespace`.

Whenever you construct an `XmlElement`, 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.

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 type-safe manner.

Whenever you construct an `XmlElement`, 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 `XmlElement`.

You can use `XmlElement` 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.

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 which of the `XmlElement` and its backing type instance supports a particular functionality. If you are using an XSD, this distinction is useful and sometimes critical.

Nillness of an element

XML has a concept of whether an element is `nil`. This state is not exactly the same as being `null`. An element that is `nil` must have no child elements, but can have attributes. Additionally, an XSD can define whether an element is nillable. A *nillable* element is allowed to be `nil`.

XMLElement constructor

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 use a more specific subclass of `AnyType`, either an XSD-based type or a simple type.

See also

- "Creating many qualified names in the same namespace" on page 264
- "Getting data from an XML element" on page 273
- "Accessing the nillness of an element" on page 280

XmlElement core property reference

To summarize, the `XmlElement` instance contains the following core properties.

`XmlElement` property	Description	Type
`QName`	A read-only property that returns the element's in Gosu XML APIs.	`QName`
`Namespace`	Returns an `XmlNamespace` object that represents the element's namespace. • If this element was created with one of the method signatures with a QName, this property returns the namespace of the QName. • If this is element is XSD-based, this property returns the namespace as defined by the XSD.	`XmlNamespace`
`TypeInstance`	Returns the element's backing type instance	`gw.xsd.w3c.xmlschema.types.complex.AnyType` or any subclass of that class
`Nillness`	Specifies whether this element is nil, which is an XML concept that is not the same as being `null`.	`boolean`

IMPORTANT If you get these properties on an XSD-based element, you must use a dollar sign prefix for the property name.

See also

- "Accessing XSD type properties" on page 257
- "Accessing the nillness of an element" on page 280

Inside an element: child elements and simple values

Separate from XML attributes and metadata, XML element can include two basic types of content:

- Child elements
- A simple value, which can represent simple types such as numbers or dates

Gosu exposes properties and methods on the XML type instances to let you access or manipulate child elements or text contents. For example, if an element is not an XSD-based element, access the properties directly, such as `element.Children`.

An element can contain either child elements or simple values, but never both at the same time. This distinction is critical 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.

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 detail is not typically necessary to remember.

See also

- "XSD-based properties and types" on page 264

Create an XmlElement

Procedure

1. To create a basic `XmlElement`, decide the element name that you want to use.
2. In a `new` expression, pass the element name to the constructor as either a `QName` object or a `String`. For example, the following line passes the name as a `String`:

   ```
   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 a more specific subclass of `AnyType`, either an XSD-based type or a simple type.
3. Optionally set attributes.
4. Optionally add child elements by calling the `addChild` method with a reference to a child element.

Example

The following Gosu code creates a new `XmlElement` without an XSD, and then adds child elements:

```
uses gw.xml.XmlElement
uses javax.xml.namespace.QName

// Create parent element
var xe = new XmlElement(new QName("http://mycompany.com/schema/vehiclexsd", "root", "veh"))

// Create child elements
var xe2 = new XmlElement(new QName("http://mycompany.com/schema/vehiclexsd", "childelement", "veh"))
xe.addChild(xe2)
var xe3 = new XmlElement("childelementWithNoNamespace") // no namespace for this element
xe.addChild(xe3)
```

```
// Print element
xe.print()
print("Namespace of xe = " + xe.$Namespace)
print("Namespace of xe2 = " + xe2.$Namespace)
print("Namespace of xe3 = " + xe3.$Namespace)
```

This code prints the following lines:

```
<?xml version="1.0"?>
<veh:root xmlns:veh="http://mycompany.com/schema/vehiclexsd">
  <veh:childelement/>
  <childelementWithNoNamespace/>
</veh:root>
Namespace of ex = {http://mycompany.com/schema/vehiclexsd}
Namespace of e2 = {http://mycompany.com/schema/vehiclexsd}
Namespace of e3 = {}
```

See also

- “XML attributes” on page 276
- “Getting data from an XML element” on page 273

Accessing XSD type properties

For some XML element properties, 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, you can 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 between properties defined on the XSD type and on the type instance that backs that type. For example, suppose the XSD defines an element’s child element named `Children`. There would unfortunately be two similar properties with the same name. Gosu prevents ambiguity by requiring properties to have a dollar-sign prefix if and only if the element is XSD-based:

- To access the child elements of an XSD-based element, use the syntax `element.$Children`.
- To access a custom child element named `Children` that the XSD defines, use the syntax `element.Children`. This name is not recommended because of the ambiguity, but Gosu can access this element correctly. 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 requirement 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 nillable elements

  Note: If you create an `XmlElement` element directly, not a subclass, the object is not an XSD type. The new object is an untyped node that uses the default type instance, which is an instance of the type `AnyType`. In such cases, there is no dollar sign prefix because there is no ambiguity between properties that are part of the type instance and properties on the XSD type.

See also

- "Accessing the nillness of an element" on page 280

Exporting XML data

The `XmlElement` class includes the following methods and properties that export XML data. All of these methods have alternative method signatures that take a serialization options object (`XmlSerializationOptions`).

Export-related methods on an XML element

Each XML element provides several methods to serialize its value.

Note: Nillable elements (`nillable="true"`) with a value of `null` are skipped and are not output. This behavior occurs even if the element is required (`minOccurs="1"`).

To ensure the correct serialization of XML elements that contain characters with high Unicode code points, best practice recommends testing serialized values that contain non-English characters.

Exporting bytes

The `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.

```
var ba = element.bytes()
```

The method has no required arguments, but can be customized by passing an optional argument of an `XmlSerializationOptions` object.

Code that sends XML with a transport that understands only character data and not byte data must always base-64 encode the bytes to compactly and safely encode binary data.

```
var base64String = gw.util.Base64Util.encode(element.bytes())
```

Base-64 encoded bytes can be decoded with the `decode` method.

```
var bytes = gw.util.Base64Util.decode(base64String)
```

Note: For example, for PolicyCenter messaging, the payload field in a `Message` entity is type `String`.

Printing to standard output stream

The `print` method serializes the element to the standard output stream (`System.out`).

```
element.print()
```

The method has no required arguments, but can be customized by passing an optional argument of an `XmlSerializationOptions` object.

Writing to a specific output stream

The `writeTo` method writes to an output stream (`java.io.OutputStream`). The output stream remains open and is not closed.

The method accepts a single required argument of an `OutputStream` object. Serialization can be customized by passing an optional second argument of an `XmlSerializationOptions` object.

Output to string with XML header

The `asUTFString` method serializes the element to a `String` object, with a header line that is compatible with UTF-8 encoding.

```
var s = element.asUTFString()
```

IMPORTANT Although the `asUTFString` method is helpful for debugging use, the `asUTFString` method is not the best way to export XML safely to external systems. Instead, it is usually best to use the XML element's `bytes` method to produce an array of bytes.

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.

The method has no required arguments. However, it can be customized by passing an optional argument for an `XmlSerializationOptions` object.

WARNING Although the `asUTFString` method adds an XML header that can specify the encoding, there is no automatic encoding transformation into bytes. To write this `String` object to another system or a file, you must be careful to do the correct encoding transformation on export. Also be aware that if you use the optional `XmlSerializationOptions` argument, you could choose another encoding. However, just like for the default `asUTFString` method signature, which assumes UTF-8 encoding, the encoding choice only affects the serialized header, not the encoding of the output bytes.

XML serialization options and methods

The `XmlSerializationOptions` type exposes properties and also 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.

These methods are chainable, which means that they return the `XmlSerializationOptions` object again. This behavior supports combining multiple method invocations in a single "chain" statement.

```
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. You can set the encoding by either of the following operations.

- 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"`.

See also

- "XML serialization options reference and examples" on page 260
- `http://tools.ietf.org/html/rfc3629`

XML serialization options reference and examples

The table to follow lists properties on a serialization options object of type `gw.xml.XmlSerializationOptions`. These properties provide serialization options:

Serialization option	Type	Description	Default value
`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 `XmlElement` 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 significant computer resources.	`true`
`XmlDeclaration`	`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 `XmlElement` objects contains no XSD-based elements, this property has no effect. GX models created in Studio cannot be invalid. As a result, they do not need to be validated when serialized to XML.	`false`
`Encoding`	`Charset`	The character encoding of the resulting XML data as a `java.nio.charset.Charset` 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: `Indent`, `LineSeparator`, `AttributeNewLine`, `AttributeIndent`.	`true`

The following properties provide options that take effect only if the `Pretty` property is `true`:

Serialization option	Type	Description	Default value
`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

XML serialization examples

The following example creates an element, then adds an element comment. Next, it demonstrates printing the element with the default settings that include comments and how to customize the output to omit comments.

```
uses gw.xml.XmlSerializationOptions

// Create an element.
var a = new 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 PolicyCenter 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.

Serialization performance and element sorting

Sorting the set of serialized objects ensures that the order of children elements of each element matches the XSD. This behavior 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 that their sequence definitions specify, you can safely disable sorting during serialization.

The element serialization sorting property (`XmlSerializationOptions.Sort`) controls whether PolicyCenter sorts the serialized XML objects.

For example, the following line of code creates an anonymous XML serialization object that performs no sorting.

```
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 example, the following 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 = docexamples.gosu.xml.demoattributes.Element1.parse(xmlDataString)
```

The following table lists the parsing methods.

Method name	Arguments	Description
`parse`	`byte[]` `byte[], XmlParseOptions`	Parse XML from a byte array with optional parsing options.
`parse`	`java.io.File` `java.io.File, XmlParseOptions`	Parse XML from a file, with optional parsing options.
`parse`	`java.io.InputStream` `java.io.InputStream, XmlParseOptions`	Parse XML from an `InputStream` with optional parsing options.
`parse`	`java.io.Reader` `java.io.Reader, XmlParseOptions`	Parse XML from a reader, which is an object for reading character streams. Optionally, add parsing options. **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: `Base64Util.encode(element.bytes())`
`parse`	`String` `String, XmlParseOptions`	Parse XML from a `String`, with optional parsing options. **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: `Base64Util.encode(element.bytes())`

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 263

Referencing additional schemas during parsing

In some advanced parsing situations, you might need to reference additional schemas as well as 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 an additional schema from an XSD, use the syntax:

```
package_for_the_schema.util.SchemaAccess
```

To see how and why you would specify additional schemas, 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 definition is analogous to saying the `ImportXSD2` type called `TypeFromSchema2` is a subclass of the `ImportXSD1` type called `TypeFromSchema1`.

The following code fails by throwing an exception because the `ImportXSD1` references the schema type `ImportXSD2:TypeFromSchema2` and Gosu cannot find that type in the current schema.

```
var schema2 = 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
// that type that the second schema defines. THIS FAILS without using the AdditionalSchemas feature.
var element = ElementFromSchema1.parse(xsdtext)
```

The `ImportXSD1` XSD type cannot locate the schema called `ImportXSD2` even though the specification extends a type that `ImportXSD2` defines.

To provide access to the additional schema, set the `AdditionalSchemas` property of the `XmlParseOptions` object to a list containing one or more `SchemaAccess` objects. The following code parses XML successfully:

```
var schema2 = 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)
```

You can remap an external namespace or XSD URL to a local XSD using the `gwxmlmodule.xml` file.

See also

- "Using a local XSD for an external namespace or XSD location" on page 287

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 `XmlElement`, 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 ex = new XmlElement(ns.qualify("localPartName"))
  ```

- Reuse the namespace of an already-created XML element. To get the namespace from an XML element instance, get its `NameSpace` property. Then, call the qualify method and pass the local part `String`:

  ```
  // 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"))
  ```

See also

- "Introduction to the XML element in Gosu" on page 253

XSD-based properties and types

The most powerful way to use XML in Gosu is to use an XSD that describes what is valid in your XML format. If you can use or generate an XSD for your XML data, it is strongly recommended that you use an XSD file. The name of each XSD file must include an `.xsd` extension in lowercase.

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:

```
PolicyCenter/modules/configuration/gsrc
```

For example, to load an XSD in the package `mycompany.schemas`, put your XSD file at the path:

```
PolicyCenter/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 more readable. You can use natural Gosu syntax to access attributes and child elements using the period operator. For example, access a child element named `ChildName` on an element with code like `element.ChildName`.

The new types are defined in a subpackage that matches the name of the XSD, except in lowercase and with no `.xsd` suffix. For example, if the XSD is at the path `gsrc/example/MyTest.xsd` and contains an element called `Address`, the new element type is `example.mytest.Address`

If you cannot use an XSD, you can use the basic properties and methods of `XmlElement` like `element.Children` and `element.getChild("ChildName")`. Getting and setting values in this way is harder to understand than using XML types and properties. This type of XML-related code that does not use XSD types is not type-safe, so Gosu cannot perform type checking, which can lead to run-time errors.

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
Element definition	Element instance
Complex type definition	Complex type instance
Simple type definition	Simple type instance
Attribute definition	Attribute 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, such as a top-level complex type.
  - It embeds a type definition inside the element definition, such as 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, you need to know which type to use. In some cases, you can do what you need in more than one way, so consider using the most readable technique.

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 also

- "XSD generated type examples" on page 268

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.

For a property, the rightmost column indicates whether the property can appear more than once, in which case the property becomes a list property. If the rightmost column contains "Yes", the property has type `java.util.List` parameterized on the type that the property has 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 is 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.

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.
- Several rows 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.

The following table lists XML definitions and the syntax for accessing an item that the definition describes.

Definition	New item	Syntax...
An element at the top level	Type	*`schema`*`.`*`ElementName`* **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 *`schema`*`.elements.`*`ElementName`*.
An element lower than the top level	Type	*`schema`*`.anonymous.elements.`*`PathFromRoot_ElementName`* For example, suppose the top level group `A` that contains an element called `B`, which contains an element called `C`. The *`PathFromRoot`* is `A_B` and the fully qualified type is *`schema`*`.anonymous.elements.`*`A_B_C`*.
A complex type at the top level	Type	*`schema`*`.types.complex.`*`TypeName`
A complex type lower than the top level	Type	*`schema`*`.anonymous.types.complex.`*`PathFromRoot`* For example, suppose a top level element `A` contains an embedded complex type. The *`PathFromRoot`* is `A`. Note that complex types defined at a level lower than the top level never have names.
A simple type at the top level	Type	*`schema`*`.types.simple.`*`TypeName`
A simple type lower than the top level	Type	*`schema`*`.anonymous.types.simple.`*`PathFromRoot`

Definition	New item	Syntax...
		For example, suppose a top level element `A` contains element `B`, which contains an embedded simple type. The path from root is `A_B`. Simple types defined at a level lower than the top level never have names.
An attribute at the top level	Type	*schema*`.attributes.`*AttributeName*
An attribute	Type	*schema*`.anonymous.attributes.`*PathFromRoot* For example, suppose a top level element `A` contains element `B`, which has the attribute C. The path*PathFromRoot* is A_B and the fully qualified type is *schema*`.anonymous.attributes.`*A_B_C*.
An attribute lower than the top level	Property	*element*`.`*AttributeName* Unlike most other generated properties on XSD types, an attribute property never transforms into a list property.

A common pattern converts a `simpleType` to `simpleContent` to add attributes to an element with a simple type. To support this pattern, Gosu provides the `ChildName` and `ChildName_elem` properties for every child element that has either a simple type or both a complex type and simple content. The property with the `_elem` suffix on its name contains the element object instance. The property without the `_elem` suffix contains the element value. Because of this design, if you later decide to add attributes to a `simpleType` element, the change does not require modification of your existing XML code.

The following table lists XML definitions for child elements and the syntax for accessing an item that the definition describes. These items can occur anywhere in the schema.

Definition	New item	Syntax...
A child element with either of: • A simple type • A complex type and a simple content	Property	*element*`.`*ChildName_elem* The property type is as follows: • If element is defined at top-level, *schema*`.`*ElementName* • If element is defined at lower levels, *schema*`.anonymous.elements.`*PathFromRoot_ElementName*. **IMPORTANT:** If this property can appear more than once, it transforms into a list type.
The value of a child element with either of: • A simple type • A complex type and a simple content	Property	*element*`.`*ChildName* The property type is as follows: • If element is defined at top-level, *schema*`.`*ElementName* • If element is defined at lower levels, *schema*`.anonymous.elements.`*PathFromRoot_ElementName*. **IMPORTANT:** If this property can appear more than once, it transforms into a list type.
A child element with a complex type and no simple content	Property	*element*`.`*ChildName* The property type is as follows: • If element is defined at top-level, *schema*`.`*ElementName* • If element is defined at lower levels, *schema*`.anonymous.elements.`*PathFromRoot_ElementName*. **IMPORTANT:** If this property can appear more than once, it transforms into a list type.

The following table lists the XML definition and the syntax for accessing an XML schema.

Definition	New item	Syntax...
Schema definition	Schema access object	*schema*`.util.SchemaAccess` This special utility object provides access to the original schema that produced this type hierarchy. This object represents this schema. Use this object if you need one schema to include another schema.

See also

- “Automatic insertion into lists” on page 270
- “Referencing additional schemas during parsing” on page 263

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, such as a numeric digit, Gosu adds an underscore prefix. If the first character is entirely invalid in a name in that context, such as a hyphen, Gosu removes the character. If after removing multiple start characters, no characters remain, Gosu uses an underscore character as the name.
- If names would be duplicates, Gosu appends numbers to remove ambiguity. For example, duplicates of the name `MyProp` become `MyProp2`, `MyProp3`, and so on.

XSD generated type examples

These examples demonstrate how to use generated types.

XSD generated type examples 1

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 and backing type instances

The following XSD defines one phone-number simple type and multiple elements using that simple type:

```
<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>
```

This XSD supports multiple ways to create and assign the phone numbers.
If you want to create three different phone numbers, use code like this:

```
var ex = new schema.Person()

ex.Cell.AreaCode = "415"
ex.Cell.MainNumber = "555-1213"

ex.Work.AreaCode = "416"
ex.Work.MainNumber = "555-1214"

ex.Home.AreaCode = "417"
ex.Home.MainNumber = "555-1215"
```

If you want to create one phone number to use in multiple elements, use code like this:

```
var ex = new schema.Person()

var p = new schema.types.complex.Phone()
p.AreaCode = "415"
p.MainNumber = "555-1212"

ex.Cell.$TypeInstance = p
ex.Work.$TypeInstance = p
ex.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 ex = new schema.Person()

var p = new schema.types.complex.Phone()
p.AreaCode = "415"
p.MainNumber = "555-1212"

ex.Cell = p // SYNTAX ERROR: cannot assign complex type instance to element type instance
ex.Work = p // SYNTAX ERROR: cannot assign complex type instance to element type instance
ex.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 ex = new schema.Person()
ex.Cell = ex.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, Gosu provides a special shorthand syntax for inserting items into the list. If you assign to the list index equal to the size of the list, the index assignment becomes an insertion.

This behavior is also true if the size of the list is zero: use the `[0]` array/list index notation and set the property. This syntax inserts the value into the list, which is equivalent to adding an element to the list. You do not have to check 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 the lists are `null`.

You use this syntax to add an element:

```
element.PropertyName[0] = childElement
```

If the list does not yet exist for a list property, Gosu creates the list when you do the first insertion.

For example, consider an element containing child elements that represent an address and the child element has the name `Address`. If the XSD declares that 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 creates intermediate XML elements as needed. Use this feature to 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 code also works with simple types derived by list (`xsd:list`) such as in 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">
          <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>
```

See also

- “XSD-based properties and types” on page 264

XSD list property example

If the possibility exists for a child element name to appear multiple times, 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>
```

Customizing XSD type code generation to exclude types

You can customize the way PolicyCenter converts XSDs to internal bytecode, which is a process called *code generation*. By default, code generation happens for every XSD type. However, for better Studio performance, you can optionally omit code generation for some XSDs in some contexts or all contexts.

To customize XSD code generation, you can modify the following file:

```
PolicyCenter/modules/configuration/res/gwxmlmodule.xml
```

This one file controls code generation for all XSDs. The file contents look similar to the following:

```
<?xml version="1.0"?>
<module xmlns="http://guidewire.com/xml/module" name="ab-customer-build">
  <dependencies>
    <dependency name="pl"/>
    <dependency name="ab"/>
  </dependencies>
  <settings>
    <setting name="gx.product" value="ab"/>
    <setting name="gx.use-gosu-enhancements" value="true"/>
  </settings>
  <excludes>
    <!-- <pattern>path/to/schema.xsd</pattern> -->
  </excludes>
  <included-only>
    <!-- <pattern>path/to/schema.xsd</pattern> -->
  </included-only>
</module>
```

Gosu supports modifying the `<excludes>` and `<included-only>` elements. All other changes are unsupported.

Exclude XSD types from code generation

Procedure

1. To find and edit the XSD code generation configuration file, in Studio, type `Ctrl+Shift+N`, and then type:

   ```
   gwxmlmodule.xml
   ```

2. For any XSDs that do not need code generation in any context, create a pattern in the `<excludes>` element:
   a. Add a `<pattern>` element to the `<excludes>` element.
   b. As content for the element, type the relative path to the XSD in the source directory including the package.

      The source directory is your `gsrc` directory for production code or `gtest` for XSDs that you need only for GUnit tests. Use forward slashes (`/`) rather than periods to separate levels in the package hierarchy.

      For example, suppose your XSD is at:

      ```
      PolicyCenter/modules/configuration/gsrc/com/mycompany/schemas/integration/rare.xsd
      ```

      Add the following `<pattern>` element to the `<excludes>` element:

```
<excludes>
  <pattern>com/mycompany/schemas/integration/rare.xsd</pattern>
</excludes>
```

3. For any XSDs that need code generation but not repeated code generation for XSD `<include>` elements, create a pattern in the `<included-only>` element:
   a. Add a `<pattern>` element to the `<included-only>` element.
   b. As content for the element, type the relative path to the XSD in the source directory including the package. Use forward slashes (`/`) rather than periods to separate levels in the package hierarchy. For example, suppose your XSD is at:

      For example, suppose your XSD is at:

      ```
      PolicyCenter/modules/configuration/gsrc/com/mycompany/schemas/integration/common.xsd
      ```

      Add the following `<pattern>` element to the `<included-only>` element:

      ```
      <include-only>
        <pattern>com/mycompany/schemas/integration/common.xsd</pattern>
      </include-only>
      ```
4. Make no other changes to the file.
5. Save the file.

Special handling of very large XSD enumerations

Due to limitations in the Java language for some types of code generation, very large enumerations are handled differently from most enumerations. If an XSD defines an enumeration (an `<xs:enumeration>` element) with more than 2000 entries, PolicyCenter converts the enumeration to a `String` value (an `<xs:string>` element).

This conversion affects your Gosu code that uses the enumeration, as well as the overall type safety of related code. In addition to changing the type of any relevant properties, the Gosu compiler cannot perform compile-time verification of the individual enumeration values. For example, if you set a property of that enumeration type, be careful to spell the value correctly. If any changes to the XSD remove an enumeration value, setting that value does not cause an automatic compilation error.

During code generation on the command line or within Studio, you will see messages that identify substitutions due to this issue:

```
*******************************************************************************
Java enum limit exceeded. Treating xs:enumeration as String
As a result there will not be static typing for this enumeration.
[gw/accelerators/test/common/xsd/test/TestLocationService_CodeExt_2013-10-17.xsd]
[gw/accelerators/test/common/xsd/test/TestLocationService_CodeExt_2013-10-17
   /types/simple/LineItemMotorcycleCategoryClosedEnumType.java]
3434 entries found
*******************************************************************************
```

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 a 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 values.

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.

See also

- "Introduction to the XML element in Gosu" on page 253

Manipulating elements and values

To get the child elements of an element, get the `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 prefix the property name with a `$`, so get the property called `$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>
```

In this XSD, note that:

- The element named `Child1` has no explicit type. The default type applies, which is `xsd:anyType`.
- The element named `Child2` has the type `xsd:int`. By definition, this element must contain an integer value. Integer is a simple type. If the integer value is empty or `null` in XML for this document, the XML is 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 the content is on an XSD type, specify the property name with the dollar sign prefix: `$SimpleValue`. To create an instance of the `XmlSimpleValue` of the appropriate type, call static methods on the `XmlSimpleValue` type with method names that start with `make`. For example, call the `makeIntInstance` method and pass it an `Integer`. It returns an `XmlSimpleValue` instance that represents an integer, and internally contains an integer. In memory, Gosu stores this information as a non-serialized value. Only during serialization of the XML, such as exporting into a byte array or using the debugging `print` method, does Gosu serialize the `XmlSimpleValue` into bytes or encoded text.
- To create simple text content, which is a text simple value, set the element's `Text` property to a `String` value. If the content is on an XSD type, specify the property name with the dollar sign prefix: `$Text`.
- If you are using an XSD, you can set the natural value in the `Value` property. If the content is on an XSD type, specify the property name with the dollar sign prefix: `$Value`. For example, use natural-looking code like `ex.$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 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. Using this syntax, you access the child element's simple value with readable code. For example, `ex.AutoCost = 5`.

Example

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 ex = new docexamples.gosu.xml.demochildprops.Element1()
var c = ex.$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 docexamples.gosu.xml.demochildprops.anonymous.elements.Element1_Child1()
ex.addChild(c1)

// Create a new CHILD element that is legal in the XSD, and add it as child.
var c2 = new 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.
ex.addChild(c2)

c = ex.$Children // Return a list of two child elements
print("Children " + c.Count + c)

print("")
ex.print()
```

This code prints the following:

```
Children 0[]

before set: null
after set with $SimpleValue: 5
after set with $Value: 7

Children 2[docexamples.gosu.xml.demochildprops.anonymous.elements.Element1_Child1
  instance, 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.

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 of these methods has an alternative signature that takes a `String`.

- `getChild(QName)` – Searches the content list for a single child with the specified `QName` name. An alternativealternative method signature takes a `String` value for the local part name. For that method signature, Gosu and 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. An alternative method signature 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, 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 = ex.getChild("Child1")

// Get a child using a QName, and "reuse" the namespace of a previous node.
var getChild2FromQName = ex.getChild(getChild1.Namespace.qualify("Child2"))

print(getChild2FromQName.asUTFString())
```

This code prints the following:

```
<?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. An alternative method signature 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. An alternative method signature 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.

See also

- “XML simple values” on page 277
- “XSD-based properties and types” on page 264
- “Collections” on page 183

XML 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 an attribute simple value by its name, specified as a `QName`. Returns a `XmlSimpleValue` object. An alternative method signature takes a `String` instead of a `QName`. This method assumes an empty namespace.
- `getAttributeValue(QName) : String` – Get attribute value by its name, specified as a `QName`. Returns a `String` object. An alternative method signature takes a `String` instead of a `QName`. This method assumes an empty namespace.
- `setAttributeSimpleValue(QName , XmlSimpleValue)` – Set an attribute simple value by its name (as a `QName`) and its value (as a `XmlSimpleValue` object). An alternative method signature takes a `String` instead of a `QName`. This method assumes an empty namespace.
- `setAttributeValue(QName , String)` – Set attribute value by its name (as a `QName`) and its value (as a `XmlSimpleValue` object). An alternative method signature takes a `String` instead of a `QName`. This method 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")
```

If you use XSDs for your elements, for typical use, do not use these methods. Instead, use the shortcuts that Gosu provides. These shortcuts provide a natural and concise syntax for getting and setting attributes.

See also

- "XML simple values" on page 277
- "XSD-based properties and types" on page 264

XML simple values

Gosu uses the `gw.xml.XmlSimpleValue` type to represent the XML format of simple values. An `XmlSimpleValue` object encapsulates a value and the logic to serialize that value to XML. Until serialization occurs, Gosu may internally store the value 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 for a `QName`

See also

- "Getting data from an XML element" on page 273

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
`makeAnyURIInstance(java.net.URI)`	Makes a URI instance

Method signature	Description
`makeBase64BinaryInstance(byte[])`	Makes a base 64 binary instance from byte array
`makeBase64BinaryInstance(gw.xml.BinaryDataProvider)`	Makes a base 64 binary instance from binary data provider
`makeBooleanInstance(java.lang.Boolean)`	Makes a boolean instance
`makeByteInstance(java.lang.Byte)`	Makes a byte instance
`makeDateInstance(gw.xml.date.XmlDate)`	Makes a date-time instance from an `XmlDate`
`makeDateTimeInstance(gw.xml.date.XmlDateTime)`	Makes a date instance from an `XmlDateTime`
`makeDecimalInstance(java.math.BigDecimal)`	Makes a decimal instance from a `BigDecimal`
`makeDoubleInstance(java.lang.Double)`	Makes a decimal instance from a Double
`makeDurationInstance(gw.xml.date.XmlDuration)`	Makes a duration instance
`makeFloatInstance(java.lang.Float)`	Makes a float instance
`makeGDayInstance(gw.xml.date.XmlDay)`	Makes a `GDay` instance
`makeGMonthDayInstance(gw.xml.date.XmlMonthDay)`	Makes a `GMonthDay` duration instance
`makeGMonthInstance(gw.xml.date.XmlMonth)`	Makes a `GMonth` instance
`makeGYearInstance(gw.xml.date.XmlYear)`	Makes a `GYear` instance
`makeGYearMonthInstance(gw.xml.date.XmlYearMonth)`	Makes a `GYearMonth` instance
`makeHexBinaryInstance(byte[])`	Makes a hex binary instance from byte array
`makeIDInstance(java.lang.String)`	Makes an `IDInstance` instance from a String
`makeIDREFInstance(gw.xml.XmlElement)`	Makes an `IDREF` instance
`makeIntegerInstance(java.math.BigInteger)`	Makes a big integer instance
`makeIntInstance(java.lang.Integer)`	Makes an integer instance
`makeLongInstance(java.lang.Long)`	Makes a long integer instance
`makeQNameInstance(javax.xml.namespace.QName)`	Makes a `QName` instance
`makeQNameInstance(java.lang.String, javax.xml.namespace.NamespaceContext)`	Makes a `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 `NamespaceContext` from an XmlElement its `NamespaceContext` property. The `String` argument is the qualified local name (including the prefix) for the new `QName`.
`makeShortInstance(java.lang.Short)`	Makes a duration instance
`makeStringInstance(java.lang.String)`	Makes a `String` instance
`makeTimeInstance(gw.xml.date.XmlTime)`	Makes a duration instance
`makeUnsignedByteInstance(java.lang.Short)`	Make unsigned byte instance
`makeUnsignedIntInstance(java.lang.Long)`	Makes an unsigned integer instance
`makeUnsignedLongInstance(java.math.BigInteger)`	Makes an unsigned long integer instance
`makeUnsignedShortInstance(java.lang.Integer)`	Makes an unsigned short integer instance

XSD to Gosu simple type mappings

For all elements that have simple types and all attributes in an XSD, Gosu creates properties based on the simple schema type of that item. 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 the supertype `anySimpleType`.

Schema type	Gosu type
anySimpleType	java.lang.String
anyURI	java.net.URI
base64Binary	gw.xml.BinaryData
boolean	java.lang.Boolean
byte	java.lang.Byte
date	gw.xml.date.XmlDate
dateTime	gw.xml.date.XmlDateTime
decimal	java.math.BigDecimal
double	java.lang.Double
duration	gw.xml.date.XmlDuration
float	java.lang.Float
int	java.lang.Integer
gDay	gw.xml.date.XmlDay
gMonth	gw.xml.date.XmlMonth
gMonthDay	gw.xml.date.XmlMonthDay
gYearMonth	gw.xml.date.XmlYearMonth
gYear	gw.xml.date.XmlYear
hexBinary	byte[]
IDREF	gw.xml.XmlElement
integer	java.math.BigInteger
long	java.lang.Long
QName	javax.xml.namespace.QName
short	java.lang.Short
time	gw.xml.date.XmlTime
unsignedByte	java.lang.Short
unsignedInt	java.lang.Long
unsignedLong	java.math.BigInteger
unsignedShort	java.lang.Integer
Any type with enumeration facets	Schema-specific enumeration type
Any type derived by list of T	java.util.List<T>
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 run time. 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

The following XSD contains defines an attribute that has restricted values.

```
<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>
```

The following code demonstrates the facet validation of the attribute that has restricted values.

```
var xml = new schema.Element1()
xml.Attr1 = 3  // Works!
xml.Attr1 = 6  // Fails with an exception exception.
```

This code prints the following:

```
gw.xml.XmlSimpleValueException: Value '6' violated one or more facet constraints
of simple type definition: value must be no greater than 5
```

Accessing the nillness of an element

XML has a concept of whether an element is `nil`. This state is not exactly the same as being `null`. An element that is `nil` must have no child elements, but can have attributes. Additionally, an XSD can define whether an element is nillable. A *nillable* 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

The following XSD contains defines an element that is nillable.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1" type="xsd:int" nillable="true"/>
</xsd:schema>
```

The following code demonstrates how to set the `$Nil` property of the element.

```
var xml = new schema.Element1()
xml.$Nil = true
xml.print()
```

This code prints the following:

```
<?xml version="1.0"?>
<Element1 xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
```

See also

- "Introduction to the XML element in Gosu" on page 253

Automatic creation of intermediary elements

If you use XSDs, whenever a property path appears in the left side of an assignment statement, Gosu creates intermediary elements to ensure that the assignment succeeds. This behavior provides a useful shortcut for typical XML coding. Use this feature to make your Gosu code significantly more understandable.

Example

The following XSD contains defines multiple nested elements.

```
<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>
```

The following code demonstrates assignment to a nested element that causes automatic creation of an instance of the intermediary element.

```
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}")
```

This code prints the following:

```
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

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: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>
```

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()
```

When 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. *Substitution group* is the standard name for the XSD substitution group hierarchy feature. 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 for this XSD feature comes from the typical usage to create a substitution group head (the group main element) with an 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.

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. However, to cast the type to the expected type for the substitution, call the `cast` method on the result with no arguments. For example:

```
xml.Address = new schema.UKAddress().cast()
```

Example

Create the following XSD file at the path `gsrc/package/groupexample.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>
```

The following code creates an `Address` property and uses the substitution group hierarchy to cast that property to a `UKAddress`:

```
var xml = new example.groupexample.Customer()
xml.Address = new example.groupexample.UKAddress().cast()
xml.print()
```

This code prints the following:

```
<?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`, which is 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. 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 use an XSD when you parse XML into an `XmlElement`, 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 1

The following XSD defines three ordered child elements for a complex XML type.

```
<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>
```

The following code assigns values to the child elements of the parent element:

```
var xml = new schema.Element1()
xml.Child2 = 2
xml.Child1 = 1
xml.Child3 = 3
xml.print()
```

This code prints the following:

```
<?xml version="1.0"?>
<Element1>
  <Child1>1</Child1>
  <Child2>2</Child2>
  <Child3>3</Child3>
</Element1>
```

Example 2

The following XSD defines two sets of ordered child elements for a complex XML type. Both sets include some of the same child elements, but use a different ordering.

```
<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>
```

The following code assigns values to the child elements of the parent element:

```
var xml = new schema.Element1()
xml.A = 5
xml.B = 5
xml.C = 5
xml.print()
print( "----------" )
xml.Q = 5
xml.print()
```

This code prints the following:

```
<?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>
```

See also

- “Automatic creation of intermediary elements” on page 281

Sorting of XSD-based elements already in the correct order

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

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>
```

The following code assigns values to the child elements of the parent element in two different orders, which are both valid:

```
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()
```

This code prints the following:

```
<?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>
```

Sorting of XSD-based elements matching multiple correct orders

If the children of an element are out of order, but multiple correct orderings exist, Gosu uses the first correct ordering defined in the schema.

Example

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>
```

The following code assigns values to the child elements of the parent element:

```
var xml = new schema.Element1()
xml.C = 5
xml.A = 5
xml.B = 5
xml.print()
```

This code prints the following:

```
<?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	`gw.xsd.w3c.soap`
SOAP envelope XSD	`gw.xsd.w3c.soap_envelope`
WSDL XSD	`gw.xsd.w3c.wsdl`
XLink XSD for linking constructs	`gw.xsd.w3c.xlink`
The XML XSD, which defines the attributes that begin with the `xml:` prefix, such as `xml:lang`	`gw.xsd.w3c.xml`
XML Schema XSD, which is the XSD that defines the format of an XSD.	`gw.xsd.w3c.xmlschema.Schema`

See also

- "The metaschema XSD that defines an XSD" on page 287

The metaschema XSD that defines an XSD

The definition of an XSD is itself an XML file. The *XML Schema* XSD is the XSD that defines the XSD format. The XML Schema XSD is also known as the *metaschema*. This schema is in the Gosu package `gw.xsd.w3c.xmlschema`. You can use the metaschema for building or parsing schemas.

Example

The following code parses the metaschema.

```
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()
```

This code prints the following:

```
<?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>
```

Gosu does not provide a way to inject a schema into the type system at run time.

Using a local XSD for an external namespace or XSD location

Sometimes an XSD refers to another XSD that is external. Requiring XSD processing to connect across the internet to access that XSD is strongly discouraged. Network connections have low performance and are unreliable in a production system, and the external XSD might not be reachable. Best practice is 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 specifying the external download location for an XSD.

In both cases, Gosu provides a registry file, `gwxmlmodule.xml`, that lets you use a local XSD instead. The file is located in the **configuration→res** module.

The default configuration includes multiple versions of `gwxmlmodule.xml` located in various modules. The **configuration→res→gwxmlmodule.xml** instance is the application-level version. The other instances of the file are for Guidewire internal use only and are not visible in Studio. Their purpose is to remap common W3C XSD files to local locations so as to avoid unnecessarily connecting to the W3C website.

See also

- "Referencing additional schemas during parsing" on page 263

Use a local XSD for an external namespace URI or XSD location URL

To remap an external namespace to a local location, edit the `gwxmlmodule.xml` file.

Procedure

1. Get a copy of the target external XSD.
2. Place that XSD in the local resources hierarchy.
3. In Studio, open the `gwxmlmodule.xml` file. The file is located in the **configuration→res** module. Add a `schemalocations` element with an `xmlns` attribute that specifies the local resource location.
4. To the `schemalocations` element, add a child `schema` element.
5. Set the element's `resourcePath` attribute to the fully qualified path to the local XSD. You must use a forward slash (`/`), not a period (`.`), to indicate any subpackage.
6. To cause Gosu to find and remap an external namespace declaration in parsed XSD, set the `xmlNamespace` attribute to the external XSD namespace URI. The value starts with `http://` so it looks like a quoted URL, but it is actually a unique ID and not a URL address.
7. To cause Gosu to find and remap an external XSD download location in parsed XSD, set the `externalLocations` attribute to the external XSD URL address. This value may also be useful for your own reference, in case you need to download the latest version of the file.

Example

Sample `schemalocations` and `schema` elements are defined in the following example.

```
<?xml version="1.0"?>
<!-- XML namespaces and external schema locations can be registered in this file to
     remap to local resources.
     The remappings are used when either an <xsd:import> is encountered without a
     <schemaLocation> element or an external 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 xmlNamespace="http://guidewire.com/archiving"
          resourcePath="gw/plugin/archiving/archiving.xsd"
          externalLocations=""/>
  <schema xmlNamespace="http://guidewire.com/importing"
          resourcePath="gw/api/importing/importing.xsd"
          externalLocations=""/>
</schemalocations>
```

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. `SchemaAccess` objects have a `Schema` property, which is the Gosu XML representation of the XSD. The `Schema` property contains the `gw.xsd.w3c.xmlschema.Schema` object that represents the XSD.

Example

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>
```

The following code prints the name of each element in the schema.

```
var schema = schema.util.SchemaAccess.Schema
schema.Element.each(\ el ->print(el.Name))
```

This code prints the following:

```
Element1
Element2
Element3
```

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))
```

This code prints the following:

```
[string, boolean, float, double, decimal, duration, dateTime, time, date, gYearMonth,
gYear, gMonthDay, gDay, gMonth, hexBinary, base64Binary, anyURI, QName, NOTATION]
```

See also

- “Parsing XML data into an XML element” on page 261
- “The metaschema XSD that defines an XSD” on page 287

Conversions from XSD types to Gosu types

The following table lists conversions that Gosu uses to convert from XSD types to Gosu types.

XSD type in custom XSD	Maps to this Gosu type
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 type in custom XSD	Maps to this Gosu type
xsd:ENTITIES	java.lang.String
xsd:ENTITY	java.lang.String
xsd:ID	java.lang.String
xsd: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:IDREF	gw.xml.xsd.IXMLNodeWithID<gw.xml.IReadOnlyXMLNode>
xsd:IDREFS	java.util.List<gw.xml.xsd.IXMLNodeWithID<gw.xml.IReadOnlyXMLNode>>
xsd:int	java.lang.Integer
xsd:integer	java.math.BigInteger
xsd:language	java.lang.String
xsd:long	java.lang.Long
xsd:Name	java.lang.String
xsd:NCName	java.lang.String
xsd:negativeInteger	java.math.BigInteger
xsd:NMTOKEN	java.lang.String
xsd:NMTOKENS	java.util.List<java.lang.String>
xsd:nonNegativeInteger	java.math.BigInteger
xsd:nonPositiveInteger	java.math.BigInteger
xsd:normalizedString	java.lang.String
xsd:NOTATION	java.lang.String
xsd:positiveInteger	java.math.BigInteger
xsd:QName	javax.xml.namespace.QName
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

chapter 21

Working with JSON in Gosu

Gosu has native support for JavaScript Object Notation (JSON) data format.

Overview of JSON support

Gosu has native support for JavaScript Object Notation (JSON) data format. JSON is an open-standard format that uses human-readable text to transmit data objects consisting of attribute-value pairs, hierarchical data structures, and arrays. Web sites often send or receive small amounts of JSON data as a lightweight alternative to the XML standard. Creation of JSON data and parsing can be implemented in any language or operating system, but is popular due to efficient web browser client-side parsing of JSON in JavaScript.

The JSON format is a binding of name-value pairs where a value is one of the following:

- A simple type, like a `String` or a number
- An object that contains name/value pairs
- An array of values

JSON itself has no inherent ability to provide type information other than those previously mentioned built-in types. For example, there is no built-in way to encode a record as a specific complex type that represents a person, a car, or a mailing address. Despite widespread use of JSON in web development, there is no standard schema for JSON. There is no XSD or DTD equivalent for JSON such that the publisher of a format can specify the correct data format to expect or validate against. As a consequence, most development environments do minimal validation and treat JSON code as untyped attribute-value pairs.

The native Gosu features called expando objects and dynamic types support the general metaphor of general purpose attribute-value pairs. You can use those Gosu features to generate JSON-like structures with a natural lightweight syntax:

```
var person: Dynamic = new() {
  :Name = "Andy Applegate",
  :Age = 39
}
```

This dynamic access can be used to get and set data in a non-type-safe way with little effort.

However, you may want to generate actual JSON data from this type of structure, or parse complex JSON data into a structure that you can naturally navigate from Gosu. Gosu provides a way of generating a type-safe class called a *structural type* from JSON data, which can assist type-safe Gosu coding with JSON data.

See also

- “Structural types” on page 241
- “Dynamic types and expando objects” on page 233
- “Dynamic access to JSON objects” on page 292
- “Structural type-safe access to JSON objects” on page 295
- “Concise dynamic object creation syntax for JSON coding” on page 297

Dynamic access to JSON objects

You can convert a JSON document as `String` data to a dynamic Gosu object. You can navigate the result using property names and array index syntax in a natural way in Gosu.

WARNING Dynamic access to JSON data is powerful and convenient but is not type safe. For example, if you misspell a property name, the Gosu editor does not catch your error at compile time.

Parsing JSON from a String object

To convert JSON data to a dynamic object with Gosu property access, use the `gw.lang.reflect.json.Json` class. Call its `fromJson` method and pass the `String` data as an argument. The method is declared to return a Java object that implements the standard interface `javax.script.Bindings`. At run time, the return type `javax.script.SimpleBindings`, which implements the `Bindings` interface.

If you declare a variable of type `Bindings` or `SimpleBindings`, you can get properties from the object. However, you must use the awkward syntax of the `get` method and passing an attribute name as a quoted `String` object.

```
var jsonParsed = Json.fromJson(jsonText)
print("Name = " + jsonParsed.get("Name"))
```

This syntax is neither convenient nor natural Gosu syntax.

Instead, Gosu natively provides natural attribute-value programming access to the `Bindings` type if you declare a variable as the type `dynamic.Dynamic`. Use this approach to get properties by name with natural syntax. Any JSON property that contains an array becomes a standard Java list, which you can use with array index syntax, collection-related Gosu enhancements, or loop syntax.

For example, run the following code in the Gosu Scratchpad:

```
uses dynamic.Dynamic
uses gw.lang.reflect.json.Json

// For this example, note that a String literal must use \" to escape quote symbols
//
//    {"Name": "Andy Applegate",
//      "Age": 29,
//      "Hobbies": ["Swimming", "Hiking"]}
//
var jsonText = "{\"Name\": \"Andy Applegate\", \"Age\": 29, \"Hobbies\": [\"Swimming\", \"Hiking\" ] }"
var j : Dynamic = Json.fromJson(jsonText)

print("Name = " + j.Name)
print("Age = " + j.Age)
print ("Number of hobbies... " + j.Hobbies.size)
for (h in j.Hobbies) {
  print(" one hobby = " + h)
}
```

This code prints the following

```
Name = Andy Applegate
Age = 29
Number of hobbies... 2
```

```
one hobby = Swimming
one hobby = Hiking
```

See also

- "Dynamic types and expando objects" on page 233

JSON APIs for URLs

Because JSON data often comes from internet requests, Gosu adds an enhancement property named `JsonContent` to the standard Java class `java.net.URL` to simplify your JSON integrations. If the `URL` object is an HTTP URL, accessing the `JsonContent` property fetches the JSON content by running the HTTP `GET` method. If the content of the `URL` object is a JSON document, this property provides a JSON object that reflects the structure and data in that document.

See also

- "Useful JSON API methods" on page 299

Parsing JSON from a URL

To get JSON data from a URL, you first create an instance of `java.net.URL` using the URL as the argument. Next, you get the `JsonContent` property on an instance of `URL`. The `JsonContent` property getter performs several complex tasks:

1. The getter makes the HTTP or other URL request from the internet and gets the result as text.
2. The getter parses the JSON text into a `Bindings` object using the static method `gw.lang.reflect.json.Json.fromJson(urlRequestString)`.
3. The getter coerces this `Bindings` object to the type `dynamic.Dynamic` before returning the object. Because the type is dynamic, you can use type inference when assigning the result to a variable.

For example, run the following code in the Gosu Scratchpad:

```
// Create a URL
var personUrl = new URL( "http://gosu-lang.github.io/data/person.json" )

// Access the URL, parse the JSON text content to a Bindings object, return as dynamic.Dynamic.
var person = personUrl.JsonContent

// Parse the dynamic object using a natural Gosu syntax
print( person.Name )
```

Note that any JSON property that contains an array becomes a standard Java list, which you can use with array index syntax or loops.

Generating URLs from JSON data

You can use JSON data to create a standard URL query. The result is a root URL, followed by a question mark, followed by a list of URL argument assignments from JSON data, separated by ampersands. The syntax is similar to:

```
http://example.com/page?arg1=val1&arg2=val2
```

Create a URL of this type using the static Gosu enhancement method `makeUrl` on the `java.net.URL` class. Pass a base URL as the first argument, and the JSON data as the second argument. The JSON data must be a bindings object like `javax.script.SimpleBindings` or anything that implements interface `javax.script.Bindings`. A Gosu expando object, such as the Gosu class `Expando`, satisfies the `Bindings` requirement.

The following example Gosu code creates JSON-like data in two ways and generates a URL that uses the JSON data for query arguments with `String` and number values. Numbers are coerced to a `String` to serialize them into text.

```
uses gw.lang.reflect.Expando
uses java.net.URL
```

```
var rootUrl = "http://example.com/en"

// ** Using Dynamic type and intialization syntax
var d: dynamic.Dynamic = new() {
  :Name = "Andy Applegate",
  :Age = 39
}
var url = URL.makeUrl(rootUrl, d)
print("Using dynamic bindings object = " + url)

// ** Using explicit Expando instantiation
var e : dynamic.Dynamic = new Expando()
e.Name = "John Smith"
e.Age = 44
url = URL.makeUrl(rootUrl, e)
print("Using dynamic expando object = " + url)
```

This code prints:

```
Using dynamic bindings object = http://example.com/en?Name=Andy+Applegate&Age=39
Using dynamic expando object = http://example.com/en?Name=John+Smith&Age=44
```

Note that space characters became + characters. This transformation is part of the process called URL encoding. Be aware that some special characters may become multiple characters as part of URL encoding.

The previous example used `Bindings` values that were `String` objects or numbers. The `makeUrl` method also supports values of other complex types:

- If the value is a `javax.script.Bindings` object, a Gosu dynamic expando object, or a list, Gosu transforms the object to JSON `String` data.
- Any other value is coerced to a `String`.

Finally, in all cases, the values are URL encoded before becoming part of the final URL.

The following example creates a new URL from a `Bindings` object with values that include one `Bindings` object and one object of list type `List<String>`:

```
uses javax.script.Bindings
uses java.net.URL

var rootUrl = "http://example.com/en"

var d: dynamic.Dynamic = new() {
  :Who = new() { :FirstName="Andy", :LastName="Applegate" },
  :Likes = {"Hiking", "Football"}
}

print("JSON of :Who = " + d.Who.toJson())
print("JSON of :Likes = " + Bindings.listToJson(d.Likes))

var url = URL.makeUrl(rootUrl, d)
print ("URL encoding = " + url)
```

The URL output for this example is harder to read because of URL encoding of special characters such as braces, comma characters, and new-line characters. Also, multiple + characters represent space characters for each line of indented JSON.

This example prints the following, which is shown formatted with added new lines in the URL for clarity:

```
JSON of :Who = {
  "FirstName": "Andy",
  "LastName": "Applegate"
}
JSON of :Likes = [
    "Hiking",
    "Football"
  ]
URL encoding = http://example.com/en
?Who=%7B%0A++%22FirstName%22%3A+%22Andy%22%2C%0A++%22LastName%22%3A+%22Applegate%22%0A%7D
&Likes=%5B%0A++++%22Hiking%22%2C%0A++++%22Football%22%0A++%5D
```

See also

- “Dynamic type” on page 233
- “Concise dynamic object creation syntax for JSON coding” on page 297.
- “Expando objects” on page 235
- “Useful JSON API methods” on page 299

Structural type-safe access to JSON objects

You can use dynamic property access with parsed JSON data, which is powerful and convenient, but not type safe. For example, if you misspell a property name, the Gosu editor cannot catch the error at compile time.

Gosu provides APIs that allow you to generate type-safe code to access JSON data. These APIs protect your code against misspelling a property name. More errors can be caught at compile time. Additionally, this approach lets you create meaningful types that you can use in the rest of your code. For example, you can create a type that represents a person, a group, or complex hierarchical business data types.

Although JSON structural types are type safe and client code validates at compile time, JSON structural types do not perform the same validations as XML schemas such as XSDs. For example, Gosu JSON structural types do not check for non-nullable fields or enforce minimum or maximum numbers of list elements. You may need to write your own validation logic to enforce your application’s unique business logic and field validation.

WARNING If property names contain unusual characters or reserved Gosu keywords, the name is changed automatically and preserved in the `@ActualName` annotation. However, the built-in APIs that convert Gosu data to JSON do not use the data that the annotation preserves.

See also

- “Dynamic access to JSON objects” on page 292

Create a JSON structural type from an example object

Procedure

1. Generate an example JSON object that contains every possible property. Every property must be non-empty and have the correct JSON type: a `String`, a number, an array, or another object.

   For example:

```
{
  "Name": "Andy Applegate",
  "Age": 29,
  "Address": {
    "Number": 123,
    "Street": "Main St",
    "City": "Foster City",
    "State": "CA"
  },
  "Hobby": [
    {
      "Category": "Sport",
      "Name": "Baseball"
    },
    {
      "Category": "Recreation",
      "Name": "Hiking"
    }
  ]
}
```

2. Upload that data as a file on a web server.
3. In the Gosu Scratchpad, use the `java.net.URL` Gosu enhancement property `JsonContent`.

```
var personUrl = new URL( "http://URL-path/file-name.json" )
```

4. In the Gosu Scratchpad, take the result of the previous step and call the `toStructure` method and pass the following two arguments:
   - A name of your new structural type, as a `String`
   - A Boolean that represents whether the structural type is mutable. If you want only property getters but not setters, pass the value `false`. If you want property getters and setters, pass the value `true`.

   For example:

```
var personUrl = new URL( "http://gosu-lang.github.io/data/person.json" )
var person: Dynamic = personUrl.JsonContent
var sj = person.toStructure( "Person", false )
```

5. Print the result, which contains Gosu code for a new structural type based on the structure of your example JSON data.

   For example:

```
print(sj)
```

   The output looks like the following:

```
structure Person {
  static function fromJson( jsonText: String ): Person {
    return gw.lang.reflect.json.Json.fromJson( jsonText ) as Person
  }
  static function fromJsonUrl( url: String ): Person {
    return new java.net.URL( url ).JsonContent
  }
  static function fromJsonUrl( url: java.net.URL ): Person {
    return url.JsonContent
  }
  static function fromJsonFile( file: java.io.File ) : Person {
    return fromJsonUrl( file.toURI().toURL() )
  }
  property get Address(): Address
  property get Hobby(): List<Hobby>
  property get Age(): Integer
  property get Name(): String
  structure Address {
    property get Number(): Integer
    property get State(): String
    property get Street(): String
    property get City(): String
  }
  structure Hobby {
    property get Category(): String
    property get Name(): String
  }
}
```

6. To use the structural type in other types, you can do either of the following:
   - To make a top-level type, copy and paste the structural type declaration into a new `.gs` file within your desired package.
   - To make an inner type, similar to an inner class in Gosu or Java, copy and paste the structural type declaration into a class declaration.

See also

- “Parsing JSON from a URL” on page 293
- “Structural type-safe access to JSON objects” on page 295
- “Problematic property names for JSON structural types” on page 296
- “Useful JSON API methods” on page 299

Problematic property names for JSON structural types

The JSON object syntax supports a wider range of characters in the name part of the name-value pairs than Gosu supports for property names. For example, space characters are permissible in JSON object name fields but not Gosu

property names. This makes it problematic to represent the structural type representation of this name as a Gosu property name. JSON also supports object name fields that would be Gosu reserved keywords such as `new` or `class`, and would cause syntax issues in type declaration or API usage.

For example, the following is legal JSON that requires special handling in Gosu structural types:

```
{
  "Space Available": "8",
  "new": "true",
  "class": 1984
}
```

For problematic property names in generated structural types, Gosu generates alternative names that satisfy Gosu property name syntax as follows:

- For illegal characters, Gosu substitutes the underscore character (_).
- For reserved keywords, Gosu reverses the case of the first character. Gosu keywords are case-sensitive, so this removes the conflict.

In both cases, Gosu preserves the original names in the annotation `@gw.lang.reflect.ActualName`.

For the previous example JSON, the generated structural type would include the following property getters and setters, with the `@ActualName` annotation as needed:

```
structure TestStructuralType {
...
  @gw.lang.reflect.ActualName( "new" )
  property get New(): String
  @gw.lang.reflect.ActualName( "class" )
  property get Class(): Integer
  @gw.lang.reflect.ActualName( "Space Available" )
  property get Space_Available(): String
}
```

WARNING If there were problematic property names due to unusual characters or reserved Gosu keywords, the name is changed automatically and preserved in the `@ActualName` annotation. However, the data preserved by the annotation is unused by the built-in APIs to convert Gosu data to JSON.

Concise dynamic object creation syntax for JSON coding

Gosu supports a concise object creation syntax similar to JSON directly in its grammar. The following Gosu features provide this functionality:

- You can omit the type name in a `new` expression if Gosu can infer the type. For example, instead of typing `new TestClass(arg1, arg2)`, you can type `new(arg1, arg2)`. The programming context already assumes the compile-time type called `TestClass`.
- You can initialize fields during instantiation using a special syntax with curly braces around a comma-delimited list of property assignments, with a colon before the property name. For example:

  ```
  new TestClass() { :Name = "John Smith" }
  ```

- Gosu supports dynamic non-type-safe access to properties and methods for variables of the type called `dynamic.Dynamic`. No declaration of specific property names or method names is required in advance. For programming contexts of type `dynamic.Dynamic`, a newly instantiated object assumes the default run-time type of `Expando`. The `Expando` class implements the `Bindings` interface. Gosu adds enhancement methods to this class to convert to JSON (`toJson`), XML (`toXml`), or Gosu object initialization code (`toGosu`).

When you combine these features, you can create objects with multiple types of data without declaring any specific structure in advance, and can convert to other formats as needed.

The following Gosu code creates a dynamic complex object and prints it in JSON syntax, XML syntax, and Gosu object initialization syntax:

```
var person: dynamic.Dynamic

person = new() {
  :Name = "John Smith",
  :Age = 39,
  :Address = new() {
    :Number = 123,
    :Street = "Main St.",
    :City = "Foster City",
    :State = "CA"
  },
  :Hobby = {
    new() {
      :Category = "Sport",
      :Name = "Swimming"
    },
    new() {
      :Category = "Recreation",
      :Name = "Hiking"
    }
  }
}

print("**** To JSON:")
print(person.toJson())

print("**** To XML:")
print(person.toXml())

print("**** To Gosu:")
print(person.toGosu())
```

This example prints the following:

```
**** To JSON:
{
  "Name": "John Smith",
  "Age": 39,
  "Address": {
    "Number": 123,
    "Street": "Main St.",
    "City": "Foster City",
    "State": "CA"
  },
  "Hobby": [
    {
      "Category": "Sport",
      "Name": "Swimming"
    },
    {
      "Category": "Recreation",
      "Name": "Hiking"
    }
  ]
}
**** To XML:
<object>
  <Name>John Smith</Name>
  <Age>39</Age>
  <Address>
    <Number>123</Number>
    <Street>Main St.</Street>
    <City>Foster City</City>
    <State>CA</State>
  </Address>
  <Hobby>
    <li>
      <Category>Sport</Category>
      <Name>Swimming</Name>
    </li>
    <li>
      <Category>Recreation</Category>
      <Name>Hiking</Name>
    </li>
  </Hobby>
</object>
```

```
**** To Gosu:
new Dynamic() {
  :Name = "John Smith",
  :Age = 39,
  :Address = new() {
    :Number = 123,
    :Street = "Main St.",
    :City = "Foster City",
    :State = "CA"
  },
  :Hobby = {
    new() {
      :Category = "Sport",
      :Name = "Swimming"
    },
    new() {
      :Category = "Recreation",
      :Name = "Hiking"
    }
  }
}
```

Note that for the Gosu initializer syntax serialization, you can evaluate the `String` data at run time to create a Gosu object:

```
var clone = eval( person.toGosu() )
```

See also

- "Optional omission of type name with the new keyword" on page 87
- "Object initializer syntax" on page 88
- "Dynamic types and expando objects" on page 233

Useful JSON API methods

The following table lists and describes some useful JSON API methods and properties.

Class	Method or property	Arguments	Description
`java.net.URL`	`JsonContent`	Not applicable	The getter for this property: • Makes the HTTP or other URL request from the internet and gets the result as text. • Parses the JSON text into a `javax.script.Bindings` object. • Coerces this `Bindings` object to the type `dynamic.Dynamic` before returning the object. For this dynamic type, Gosu uses type inference when assigning the result to a variable. For example: `var person: Dynamic = personUrl.JsonContent`
`gw.lang.reflect.json.Json`	`fromJson`	`String`	Parses JSON data as a `String` and converts it to the structural type. To use natural Gosu syntax for properties of the result, assign the result to a variable of type `dynamic.Dynamic`. For example: `var j : Dynamic = Json.fromJson(jsonText)`
	`fromJsonFile`	`java.io.File`	Get JSON data from a `File` object and convert it to the structural type. To use natural Gosu syntax for properties of the result, assign the result to a variable of type `dynamic.Dynamic`.

Class	Method or property	Arguments	Description
	`fromJsonUrl`	`String` or `java.net.URL`	Gets JSON data from a URL and convert it to the structural type. One method signature takes a URL as a `String`, and one method signature takes the URL as a `java.net.URL` object. To use natural Gosu syntax for properties of the result, assign the result to a variable of type `dynamic.Dynamic`.
`dynamic.Dynamic`	`toGosu`	No arguments	Converts an instance of the structural type to Gosu code that creates that instance. The return value is a `String` object containing Gosu code. Escaped Unicode characters in the structural type remain unchanged in the Gosu code.
	`toJson`	No arguments	Converts an instance of the structural type to JSON. The return value is a `String` object containing JSON data. Escaped Unicode characters in the structural type remain unchanged in the JSON data.
	`toXml`	No arguments	Converts an instance of the structural type to XML. The return value is a `String` object containing XML data. Escaped Unicode characters in the structural type are converted to unescaped Unicode characters in the XML data.

chapter 22

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 calculates and generates text in the location in which the code appears in the template text.

Template overview

Templates are text with embedded Gosu code inside a larger block of text. The embedded Gosu code optionally calculates and generates text at the location of the code in the template text.

Gosu provides the following mechanisms to use a Gosu template:

- **Template syntax inside a text literal** – Inside your Gosu code, use template syntax for an in-line `String` literal value containing an embedded Gosu expression. 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 produce dynamic output at run time as a `String` value.
- **Scriptlet syntax inside a text literal** – Inside your Gosu code, use scriptlet syntax for in-line Gosu statements.
- **Separate template file** – Define a Gosu template as a separate file that you can execute from other code to perform an action and generate output. If you use a separate template file, you can use additional features such as passing custom parameters to your template.

See also

- "Using template files" on page 305

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 myVariable = "One plus one equals " + mycalc + "."
```

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

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

If you print this variable, Gosu prints:

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

Gosu evaluates 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`.

Alternative template expression syntax <%= ... %>

The syntax `${ EXPRESSION }` is the preferred style for template expressions.

Gosu also 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 myVariable = "One plus one equals <%= 1 + 1 %>."
```

Running Gosu statements in a template scriptlet

Text enclosed with the text `<%` and `%>` evaluates at run time in the order the parser encounters the text but generates no output from the result. These character sequences are called scriptlet tags. Note that this type of tag has no equal sign in the opening tag. *Scriptlet tags* enclose Gosu statements, not expressions. Gosu evaluates all plain text between scriptlet tags within the scope and the logic of the scriptlet code.

The scriptlet tags are available in template files and in `String` literals that use template syntax.

The following 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 }
```

The result of this template is the following:

```
One plus two is 3
```

The result of the scriptlet tag at the beginning of the template does not generate any output. The statement inside the scriptlet tags creates a variable `MyCalc`, and assigns the result of a calculation to that variable. The subsequent expression uses the expression delimiters `${` and `}` to cause Gosu to retrieve the value of the variable `MyCalc`, which is `3`.

The scope of the Gosu code 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 result of the following template code is “x is 5” if the variable `MyCalc` has the value 5, otherwise the result is “MyCalc is not 5”:

```
<% if (MyCalc == 5) { %>
MyCalc is 5
<% } else { %>
MyCalc is not 5
<% } %>
```

The `if` statement controls the flow of execution of later elements in the template. You use this feature 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.

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 template might generate output such as:

```
Name: John Smith
Name: Wendy Wheathers
Name: Alice Applegate
```

The following code shows the use of scriptlets in a `String` value:

```
var str = "<% for (i in 1..5) { var odd = (i % 2 == 1) %>${i} is ${ (odd?"odd":"even") }\n<% } %>"
print(str)
```

This code prints:

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
```

IMPORTANT Gosu has 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.

See also

- "Template parameters" on page 307

Escaping special characters for templates

Gosu templates use standard characters 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 escape special characters.

For non-template-tag use, escape ${ or <%

Gosu templates use the following two-character sequences to begin a template expression

- `${`
- `<%`

In a `String` literal in your code, if you want to use these sequences to indicate template tags, you do not need to escape these special characters.

If you require either of those two special two-character sequences in your `String`, rather than as a template tag, escape the first character of that sequence. To escape a character, add a backslash character immediately before the character to escape. 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 your `String` uses the initial character of the template sequence, but the next character does not complete that sequence, you do not need to escape the character. For example:

- To define a variable containing the non-template text `"Hello$There"`, just use:

```
var s = "Hello$There"
```

- To define a variable containing the non-template text `"Hello<There"`, just 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 any quotation marks with the syntax `\"` to avoid ambiguity about the end of the `String`. For example:

```
var quotedString = "\"This string has quotation marks around it\", is that correct?"
```

This code creates a `String` with the following value, including quotation marks:

```
"This string has quotation marks around it", is that correct?
```

If you use a template, this rule does not apply between your template-specific opening and closing tags that contain Gosu code. Instead, use standard Gosu syntax for the code between those open and closing tags.

For example, the following two lines are valid Gosu code:

```
var s1 = "${ "1" }"
var s2 = "${ "1" } \"and\" ${ "1" }"
```

Note that the first character within the template's Gosu block is an unescaped quotation mark.

The following code is invalid because of incorrect escaping of internal quotation marks:

```
var s = "${ \"1\" }"
```

In this invalid case, the first character in the template's Gosu block is an escaped quotation mark.

In the rare case that your Gosu code requires creating a `String` literal containing a quotation mark character, remember that the standard Gosu syntax rules apply. You need to escape any quotation marks inside the `String` literal. For example, the following is valid Gosu:

```
var quotedString = "${ "\"This string has quotation marks around it\", is that correct?" }"
```

Note that the first character in the template's Gosu block is an unescaped quotation mark. This template generates a `String` with the value:

```
"This string has quotation marks around it", is that correct?
```

IMPORTANT Be careful with how you escape quotation mark 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. In addition, template files provide 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 syntax with the template expression syntax, which requires Gosu expressions rather than Gosu statements. The results of scriptlet tags generate no output.
- **Define template parameters** – Template files can define parameters that you pass to the template at run time.
- **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 by using the `extends` feature.

See also

- "Template overview" on page 301
- "Running Gosu statements in a template scriptlet" on page 302

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. The template is a first-class object in the Gosu type system within its package namespace.

In your template file, include the template body with no surrounding quotation marks. The following line is a simple template:

```
One plus one equals ${ 1 + 1 }.
```

To create a new template within Studio, in the **Project** pane within the **gsrc** section, right-click a package. Next, click **New→Gosu Template**. In the **New Gosu Template** dialog, type a meaningful name, and then click **OK**.

Rendering to a String

To run a template, get a reference to your template and call the `renderToString` method of the template. The `renderToString` method returns the template results as a `String` value.

For example, suppose you create a template file `NotifyAdminTemplate.gst` within the package `mycompany.templates`. The fully qualified name of the template is `mycompany.templates.NotifyAdminTemplate`.

Use the following code to render your template:

```
var x = mycompany.templates.NotifyAdminTemplate.renderToString()
```

The variable `x` contains the `String` output of your template.

If you need to pass template parameters to your template, provide arguments to the `renderToString` method.

Rendering to a writer

Optionally, you can render the template directly to a Java writer object. Your writer must be an instance of `java.io.Writer`. Get a reference to the template and call its `render` method. Pass the writer as an argument to the `render` method.

For example, suppose you create a template file `NotifyAdminTemplate.gst` within the package `mycompany.templates`. If the 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, put your additional parameters after the writer argument.

See also

- "Template parameters" on page 307

Create and use a template file

About this task

This example 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. This example produces static text. A typical template uses one or more parameters to produce dynamic output.

Procedure

1. Using Studio, create a new template. In the **Project** pane within the **gsrc** section, right-click a package. Next, click **New→Gosu Template**. In the **New Gosu Template** dialog, type a meaningful name, and then click **OK**.

   For example, the fully qualified name of your template could be `mycompany.templates.IntegerStringsInArray`.
2. Use a template scriptlet to create a static array or other iterable object:
   a. Use `<%` to begin your definition and type normal Gosu code to define the object.

   ```
   <% var MyArray : ArrayList<String> = {
     "1",
     "Two",
     "3.0"
   }
   ```

   b. End the scriptlet, using `%>`.

   Alternatively, define a parameter that specifies an array or other iterable object.
3. Optionally, generate some static text.

   For example:

   ```
   Testing strings for number content:
   ```
4. Use a template scriptlet to iterate over your object. For example:
   a. Use `<%` to begin your loop. For example:

   ```
   <% for (var obj in MyArray) {
   ```

   b. Before ending the scriptlet, set up a variable with your data to export and optionally perform application logic.

   ```
   var string = obj.toString()
   try {
     obj.toDouble()
   ```

c. End the scriptlet, using `%>`.

5. Insert a template expression to export the value of your variable, by enclosing a Gosu expression with `${` and `}` tags. Use your variable and, optionally, static text to produce data to export. Use more scriptlet tags as necessary to produce valid Gosu code.

```
${ string } is a number<%} catch (e : Exception) { %>
"${ string }" is not a number
```

6. Optionally, generate some static text.
7. Add another template scriptlet by using `<%` 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.

```
  <% }
}%>
```

8. If Studio is running a server, you must load the changed file by remaking the project. From the menu, click **Build→Make Project**.
9. In Gosu Scratchpad, run the template by using code like the following:

```
print(mycompany.templates.IntegerStringsInArray.renderToString())
```

Result

Running this code produces text like the following:

```
1 is a number
"Two" is not a number
3.0 is a number
```

Example

The complete code for this template file example looks like:

```
<% var MyArray : ArrayList<String> = {
  "1",
  "Two",
  "3.0"
}
%>
Testing strings for number content:
<% for (var obj in MyArray) {
  var string = obj.toString()
  try{
    obj.toDouble() %>
    ${ string } is a number<%} catch (e : Exception) { %>
    "${ string }" is not a number
  <% }
}%>
```

See also

- “Template parameters” on page 307

Template parameters

You can pass parameters of any type to your self-contained Gosu template files. The syntax for defining parameters for a template is:

```
<%@ params(ARGLIST) %>
```

ARGLIST is an argument list such as for a standard Gosu function.

You can use template parameters in template files, but not in `String` literals that use template syntax. In a template file, you can use a parameter in either the template expression syntax (`${` and `}`) or template scriptlet syntax (`<%` and `%>`). The expression syntax always returns a result and generates additional text. The scriptlet syntax executes Gosu statements.

For example, suppose you create a template file `NotifyAdminTemplate.gst` within the package `mycompany.templates`. Edit the file to contain the following lines of code:

```
<%@ 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 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.

To run your template, you use 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)
```

IMPORTANT Gosu has 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.

Support and use parameters in a template file

Procedure

1. Create a template file, for example, `mycompany.templates.NotifyAdminTemplate.gst`.
2. At the top of the template file, create a parameter definition.

   For example, the following argument list includes a `String` argument and a `boolean` argument:

   ```
   <%@ params(personName : String,
   contactHR: boolean) %>
   ```
3. Later in the template, use template tags that use the values of those parameters.

   For example:

   ```
   The person ${ personName } must update their contact information in the company directory.

   <% if (contactHR) { %>
     Call the human resources department immediately.
   <% } %>
   ```
4. If Studio is running a server, you must load the changed file by remaking the project. From the menu, click **Build→Make Project**.
5. To run the template, add your parameters to the call to the `renderToString` method or after the writer parameter to the `render` method.

   For example, in the Gosu Scratchpad:

```
print(mycompany.templates.NotifyAdminTemplate.renderToString("Pamela Castillo", true))
```

Maximum file-based template size

The Gosu compiler converts a Gosu template file into generated Java class files. The Java compiler has a maximum size of 65535 bytes for any class method. Sufficiently large Gosu templates can result in templates that fail at run time due to this JVM limitation.

If you have very large templates, break them into nested templates. For example, suppose you have a large template that generates three separate sections of a large page. Create three additional templates that each generate one part of the content. The original template could contain code that calls the other three templates. This design practice prevents breaching the size limit. Additionally, this style produces more readable and more manageable code.

Extending a template from a class

Gosu provides a special syntax to simplify calling static methods on a class. 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.

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.

You can use the `extends` syntax in template files, but not in `String` literals that use template syntax.

Example

Suppose your template needs 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.mailTemplate %>
Hello! The email address is ${sanitizeEmailAddress(address)}
```

Notice that the class name does not appear immediately before the call to the static method.

See also

- “Modifiers” on page 157

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 --%>
```

You can use template comments in template files, but not in `String` literals that use template syntax.

For example:

```
My name is <%-- this is a comment --%>John.
```

Rendering this template file produces:

```
My name is John.
```

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 invalidates the XML or confuses the recipient of the XML. For instance, beware of unescaped "`<`" or "`&`" characters in a notes field. If possible, export data within an XML `<CDATA>` tag, which allows more types of characters and character strings without problems of unescaped characters.

chapter 23

Query builder APIs

The query builder APIs let you retrieve information from PolicyCenter application databases. The API framework models features of SQL `SELECT` statements to make object-oriented Gosu queries.

Overview of the query builder APIs

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

The processing cycle of a query

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

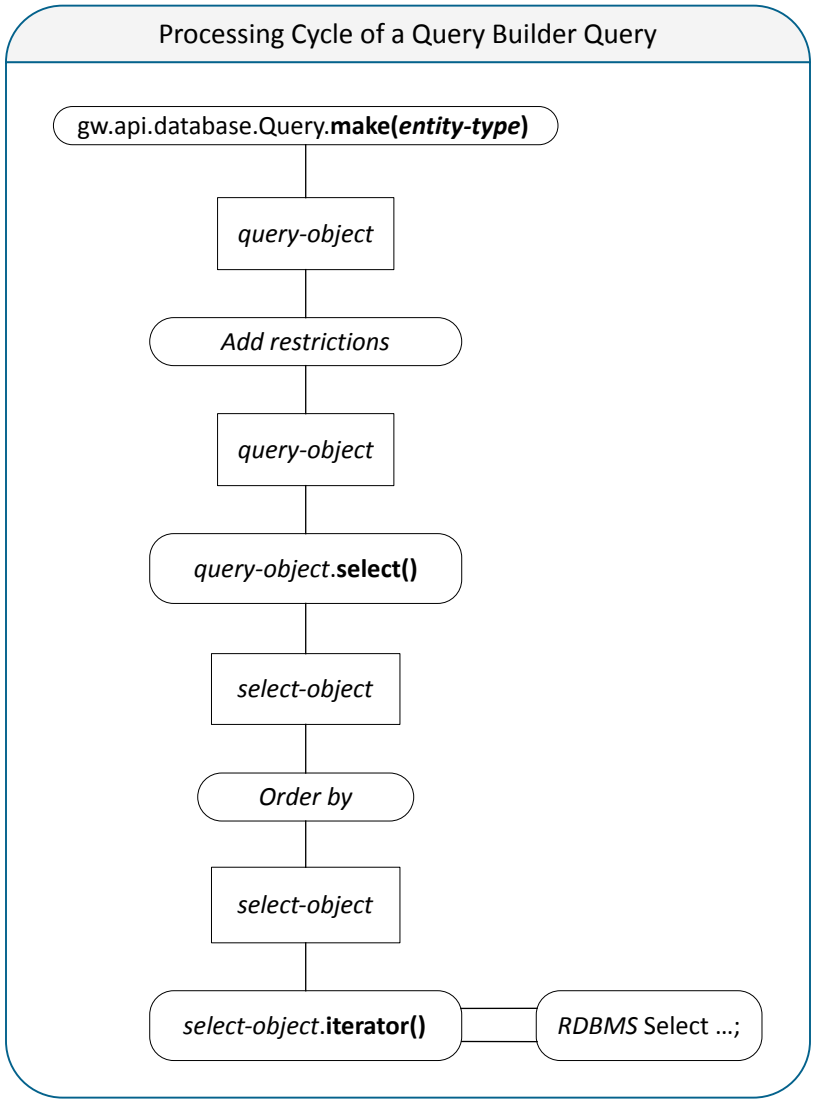
A query object
Specifies which PolicyCenter entity instances to fetch from the application database

A select object
Specifies how to order and group selected entity instances

The processing cycle of a PolicyCenter 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 select object.
4. Refine your select object by ordering the selected items.
5. Iterate your select 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.



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, the query is not performed until you start to access the result set. The application database orders the results while fetching data. Any action that you take on result objects to return information, such as getting result counts or starting to iterate the result, triggers query execution in the application database.

Comparison of SQL select statements and query builder APIs

The query builder APIs support many, but not all, features of the SQL `SELECT` statement.

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 example	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.
`FROM table`	`var query = Query.make(entity-type)`	Declares the primary source of data.
`DISTINCT`	`query.withDistinct(true)`	Eliminates duplicate items from the results.

Keyword or clause in SQL	Equivalent in query builder APIs example	Purpose
column1 AS A[, *column2* AS B [...]]	query.select({ QuerySelectColumns.pathWithAlias("A", Paths.make(entity#*field1*)) [, QuerySelectColumns.pathWithAlias("B", Paths.make(entity#*field2*)) ...] })	Produces a result that contains a set of name/value pairs, instead of a set of entity instances.
JOIN *table* ON *column*	var table = query.join(entity#*foreign-key*)	Joins a dependent source of information to the primary source, based on a related column or field.
WHERE	query.compare(entity#*field-name*, *parameters*) query.compareIgnoreCase(entity#*field-name*, *parameters*) query.between(entity#*field-name*, *parameters*) query.compareIn(entity#*field-name*, *parameters*) query.compareNotIn(entity#*field-name*, *parameters*) query.startsWith(entity#*field-name*, *parameters*) query.contains(entity#*field-name*, *parameters*)	Fetches information that meets specified criteria.
ORDER BY *column1*[, *column2*[...]	var orderedResult = result.orderBy(QuerySelectColumns.path(Paths.make(entity#*field1*)) [.thenBy(QuerySelectColumns.path(Paths.make(entity#*field2*))) [...]]	Sort results by specific columns or fields.
GROUP BY	Implied by aggregate functions on fields	Return results grouped by common values in a field.
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. **Note:** You cannot use the union method on query objects passed as arguments to the subselect method.
FIRST	result.FirstResult	Limit the results to the first row, after grouping and ordering.
TOP | LIMIT	result.getCountLimitedBy(*int*) result.setPageSize(*int*)	Limit the results to the first *int* number of rows at the top, after grouping and ordering. These query builder API methods provide similar functionality in a portable way.

Unsupported SQL features in query builder APIs

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

Key-word or clause in SQL	Meaning in SQL	Why the APIs do not support it
`FROM table1, table2[, ...]`	Declares the tables in a join query, beginning with the primary source of information on the left and proceeding with dependent sources of information towards the right.	• 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.
`EXCEPT`	Removes items fetched by a query that are in the results fetched by a second query.	This SQL feature is seldom used.
`INTERSECT`	Reduce items fetched by two separate queries to those items in both results only.	Query intersection often causes a performance problem. A better choice is to use a more efficient query type. For example, if both sides of the `INTERSECT` clause query the same table, use a single query. Use an `AND` operator to combine the restrictions from both sides of the `INTERSECT` to restrict the result.

See also

- "Building a simple query" on page 314
- "Making an inner join with the foreign key on the right" on page 337
- "Working with row queries" on page 347
- "Joining a related entity to a query" on page 334
- "Restricting a query with predicates on fields" on page 317
- "Ordering results" on page 363
- "Applying a database function to a column" on page 350
- "Accessing the first item in a result" on page 366
- "Setting the page size for prefetching query results" on page 373
- "Determining if a result will return too many items" on page 367

Building a simple query

Consider a simple query in SQL that returns 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.

The following Gosu code constructs and executes a functionally equivalent query to the preceding SQL example.

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

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

In response, the PolicyCenter 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 meets any restrictions that you added. Calling the `iterator` method in the preceding Gosu code causes the application database to fetch all `Address` instances from the application database into the result object.

The query builder API can use a view entity as the primary entity type in the same way as a standard entity. A view entity can provide straightforward access to a subset of columns on primary and related entities.

See also

- *Configuration Guide*

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.

The following Gosu code constructs and executes a functionally equivalent query to the preceding SQL example.

```
uses gw.api.database.Query

var query = Query.make(Address)
query.compare(Address#City, Equals, "Chicago")
var select = query.select()
var result = select.iterator() // Execute the query and return an iterator to access the result.
```

In response to the preceding Gosu 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 can 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;
```

The following Gosu code sorts the instances in the result object in the same way as the preceding SQL example.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns // Import the class that defines a column
uses gw.api.path.Paths                  // Import the class that builds a path to a column

var query = Query.make(Address)
query.compare(Address#City, Equals, "Chicago")
var select = query.select()
// Specify to sort the result by postal code.
select.orderBy(QuerySelectColumns.path(Paths.make(Address#PostalCode)))
```

See also

- "Ordering results" on page 363

Accessing the results of a simple query

You use queries to fetch information from a database, and then work with the results. You embed SQL queries in your application code so you can bind application data to the SQL queries and submit them programmatically to the PolicyCenter relational database. Result sets that relational databases return provide a programmatic cursor to let you access each result row sequentially.

You use the query builder APIs to embed queries naturally in your Gosu application code. You bind application data to your queries and submit the queries programmatically with standard Gosu syntax. PolicyCenter application databases return select objects that implement the `java.lang.iterable<T>` interface to let you access each entity instance in the result sequentially.

For example, the following Gosu 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
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

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

// Specify how you want the result returned.
var select = query.select()
select.orderBy(QuerySelectColumns.path(Paths.make(Address#PostalCode)))

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

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

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

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

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

// Specify how you want the result returned.
var select = query.select()
select.orderBy(QuerySelectColumns.path(Paths.make(Address#PostalCode)))

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

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

Restricting a query with predicates on fields

In most cases, you do not need to process all rows in a table. Typically, you need to restrict the set of rows, for example, by date or geographic location.

A *predicate* is a condition that selects a subset of values from a larger set. Predicates are independent of the sets to which you apply them. You can create predicates to apply to multiple sets of values to select subsets of those values. For example, the expression “is male” is a predicate that 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 male people.

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 methods on query objects to apply predicates to the sets of values in specific entity fields.

See also

- “Restricting query results by using fields on joined entities” on page 343
- “Restricting query results with fields on primary and joined entities” on page 345

Using a comparison predicate with a character field

The result of comparing the value of a character field with another character value differs depending on the language, search collation strength, and database that your PolicyCenter uses. For example, case-insensitive comparisons produce the same results as case-sensitive comparisons if PolicyCenter has a linguistic search strength of primary.

PolicyCenter converts character literals that you specify in query builder predicates to bind variables in the SQL statement that PolicyCenter sends to the relational database.

See also

- *Globalization Guide*

Case-sensitive comparison of a character field

In SQL, you apply comparison 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 methods on the query object. The following Gosu code applies the `compare` method 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(Company#Name, Equals, "Acme Rentals")

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

In the result set, the values in the character field exactly match the comparison value that you provide to the `compare` method, including the case of the comparison value. For example, the preceding query builder code selects only “Acme Rentals,” but not “ACME Rentals.”

Note: In configurations that use primary or secondary linguistic sort strength, the preceding query builder code does not perform a case-sensitive comparison. If either of your PolicyCenter or the relational database that your PolicyCenter uses have these configurations, the result includes both "Acme Rentals" and "ACME Rentals."

See also

- "Comparing a column value with a literal value" on page 327
- "Comparing a typekey column value with a typekey literal" on page 328
- *Globalization Guide*

Case-insensitive comparison of a character field

The case of letters can cause a problem when you compare character values. You might not know the case of individual characters in the field values that you want to select. For example, you want to select the company named "Acme Rentals." There is only one instance of the `Company` entity type with that name. However, you do not know whether the company name in the database is "Acme Rentals," "ACME RENTALS," or even "acme rentals."

SQL `WHERE` clauses let you apply case-insensitive comparisons with functions that convert values in columns to upper or lower case before making a comparison. 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 `compareIgnoreCase` method for case-insensitive comparisons, as the following Gosu code shows.

```
uses gw.api.database.Query

var query = Query.make(Company)
query.compareIgnoreCase(Company#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.

See also

- *Globalization Guide*

Comparison of a character field to a range of values

In SQL, you use the `BETWEEN` keyword to apply comparison predicates to an inclusive range of column values in the `WHERE` clause. For example, the following SQL statement applies the predicate `BETWEEN "Bank" AND "Business"` to values in the `name` column of the `companies` table. The query returns all companies with names between the values "Bank" and "Business," including any company that has the name "Bank" or the name "Business."

```
SELECT * from companies
  WHERE name BETWEEN "Bank" AND "Business";
```

With the query builder APIs, you use the `between` method to apply a range predicate to entity fields. The following Gosu code applies the `between` method to values in the `Name` field of `Company` entity instances to select companies with names between the values "Bank" and "Business."

```
uses gw.api.database.Query

// Query the Company instances for a range of company names.
var query = Query.make(Company)
query.between(Company#Name, "Bank", "Business")

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

The `between` method performs a case-sensitive comparison of the values in the character field to the comparison values. For example, the preceding query builder code selects only “Building Renovators,” but not “building renovators.”

Note: In configurations that use primary or secondary linguistic sort strength, the preceding query builder code does not perform a case-sensitive comparison. If either of your PolicyCenter or the relational database that your PolicyCenter uses have these configurations, the result includes both “Building Renovators” and “building renovators.”

To perform a case-insensitive comparison on a range of values for a character field, use the `compareIgnoreCase` method, as the following Gosu code shows.

```
uses gw.api.database.Query

// Query the Company instances for a range of company names.
var query = Query.make(Company)
query.compareIgnoreCase(Company#Name, GreaterThanOrEquals, "bank")
query.compareIgnoreCase(Company#Name, LessThanOrEquals, "business")

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

See also

- “Handling of null values in a range comparison” on page 325

Partial comparison from the beginning of a character field

Sometimes you need to make a partial comparison that matches the beginning of a characters field, instead of matching the entire field. For example, you want to select a company named “Acme”, but you do not know whether the full name is “Acme Company”, “Acme, Inc.”, or even “Acme”.

In SQL, you apply a partial comparison predicate with the `LIKE` operator in the `WHERE` clause. The following SQL statement applies the predicate `LIKE "Acme%"` to values in the `name` column of the `companies` table. The percent sign (`%`) is an SQL wildcard that matches zero or more characters.

```
SELECT * from companies
  WHERE name LIKE "Acme%";
```

The query result contains companies with names that begin with “Acme”.

With the query builder APIs, you apply partial comparison predicates that match from the beginnings of character fields by using the `startsWith` method on the query object. Test the use of the `startsWith` method in a realistic environment. Using the `startsWith` method as the most restrictive predicate on a query can cause a delay on the user interface.

The following Gosu code applies the `startsWith` predicate method to values in the `Name` field on `Company` entity instances.

```
uses gw.api.database.Query

// Query the Company instances for a specific company.
```

```
var queryCompany = Query.make(Company)
queryCompany.startsWith(Company#Name, "Acme", false)

// Fetch the data with a for loop.
var result = queryCompany.select()
for (company in result) {
  print (company.Name)
}
```

The sample code prints the names of companies that begin with “Acme”.

Note: The value `false` as the third argument of the `startsWith` method specifies case-sensitive matching of values. In configurations that use primary or secondary linguistic sort strength, the preceding query builder code does not perform a case-sensitive comparison. If either your PolicyCenter application or the relational database that your PolicyCenter uses have these configurations, the result includes the names of companies that begin with both “Acme” and “ACME.”

See also

- “Case-insensitive partial comparisons of a character field” on page 321
- *Globalization Guide*

Partial comparison of any portion of a character field

Sometimes you need to make a partial comparison that matches any portion of a character field, not just from the beginning of the field. For example, you want to select an activity that is a review, but you do not know the type of review. In this case, you want to search for “Review” anywhere within the field for the subject of the activity.

In SQL, you apply partial comparison predicates with the `LIKE` operator in the `WHERE` clause. The following SQL statement applies the predicate `LIKE "%Review%"` to values in the `Subject` column of the `pc_activity` table. The percent sign (`%`) is an SQL wildcard that matches zero or more characters. The query also restricts the set of activities to those that have a particular assigned user. Using the `LIKE` operator without another, more restrictive predicate causes the database to do a full table scan because no index is available to the query.

```
SELECT * FROM pc_activity
  INNER JOIN pc_user
    ON pc_user.ID = pc_activity.AssignedUserID
  WHERE pc_activity.Subject LIKE "%Review%"
    AND pc_user.PublicID = "pc:8";
```

The query result contains activities with a subject that includes “Review” and that have a particular assigned user.

The following query uses the `LIKE` operator in the `WHERE` clause and causes a full table scan because there is no other restriction on the primary table:

```
SELECT * FROM pc_activity
  WHERE pc_activity.Subject LIKE "%Review%";
```

With the query builder APIs, you apply partial comparison predicates that match any portion of character fields by using the `contains` method on the query object. Test the use of the `contains` method in a realistic environment. Using the `contains` method as the most restrictive predicate on a query causes a full-table scan in the database because the query cannot use an index.

WARNING For a query on a large table, using `contains` as the most restrictive predicate can cause an unacceptable delay to the user interface.

The following Gosu code applies the `contains` predicate method to values in the `Subject` field on the `Activity` instance for which a `User` instance is the assigned user.

```
uses gw.api.database.Query

// Query the Activity instances for a particular user based on subject line.
var queryActivity = Query.make(Activity)
```

```
// Join User as the dependent entity to the pimary entity Activity.
queryActivity.join(Activity#AssignedUser).compare(User#PublicID, Equals, "pc:105")

// Add a predicate on the primary entity.
// The value "false" means "case-senstive."
queryActivity.contains(Activity#Subject, "Review", false)

// Fetch the data with a for loop.
for (activity in queryActivity.select()) {
  print("Assigned to " + activity.AssignedUser + ": " + activity.Subject)
}
```

This Gosu code prints the assigned user and the subject of the activity where the subject of the activity contains "Review" and the assigned user has a particular public ID.

Note: The value `false` as the third argument of the `contains` method specifies case-sensitive matching of values. In configurations that use primary or secondary linguistic sort strength, the preceding query builder code does not perform a case-sensitive comparison. If either your PolicyCenter application or the relational database that your PolicyCenter uses have these configurations, the result includes the subjects of activities that contain both "Review" and "review."

See also

- "Case-insensitive partial comparisons of a character field" on page 321

Case-insensitive partial comparisons of a character field

Sometimes you need to make partial comparisons in a case-insensitive way. In SQL, you apply case-insensitive partial comparison predicates with functions that convert the column values to upper or lower case before making the comparison. The following SQL statement converts the values in `Subject` to lower case before comparing them to `review`.

```
SELECT * FROM pc_activity
  INNER JOIN pc_user
    ON pc_user.ID = pc_activity.AssignedUserID
  WHERE LOWER(pc_activity.Subject) LIKE "%review%"
    AND pc_user.PublicID = "pc:8";
```

With the query builder APIs, you use the third parameter of the `startsWith` and `contains` methods to specify whether to make the comparison in a case-insensitive way.

- `true` – Case-insensitive comparisons
- `false` – Case-sensitive comparisons

IMPORTANT In configurations using primary or secondary linguistic sort strength, query builder code does not perform a case-sensitive comparison even if you use a value of `false` for the third parameter.

The following Gosu code applies the `contains` predicate method to values in the `Subject` field on the `Activity` instance for which a `User` instance is the assigned user. The code requests a case-insensitive comparison with the value `true` in the third parameter.

```
uses gw.api.database.Query

// Query the Activity instances for a particular user based on subject line.
var queryActivity = Query.make(Activity)

// Join User as the dependent entity to the pimary entity Activity.
queryActivity.join(Activity#AssignedUser).compare(User#PublicID, Equals, "pc:105")

// Add a predicate on the primary entity.
// The value "true" means "case-insenstive."
queryActivity.contains(Activity#Subject, "Review", true)

// Fetch the data with a for loop.
for (activity in queryActivity.select()) {
  print("Assigned to " + activity.AssignedUser + ": " + activity.Subject)
}
```

If you choose case-insensitive partial comparisons, Gosu generates an SQL function that depends on your PolicyCenter and database configuration to implement the comparison predicate. If the data model definition of the column sets the `supportsLinguisticSearch` attribute to `true`, Gosu uses the denormalized version of the column instead.

See also

- *Globalization Guide*

SQL wildcard characters for partial character comparisons

SQL supports the following wildcard characters in the `LIKE` operator:

% (percent sign)
Represents zero, one, or more characters

_ (underscore)
Represents exactly one character

The query builder API does not support the use of SQL wildcard characters in the `startsWith` or `contains` predicate methods. If you include wildcard characters in the search-term parameter of either method, the query builder methods escape their special meaning by preceding them with a backslash (\).

Using a comparison predicate with a date and time field

Different brands of database provide different functions to work with timestamp columns in SQL query statements. The query builder APIs offer the following database functions for working with timestamp columns on any supported database:

- `DateDiff` – Computes the interval between two date and time fields
- `DatePart` – Retrieves the parts of a date-and-time field
- `DateFromTimestamp` – Retrieves the date part of a date-and-time field

Comparing the interval between two date and time fields

Use the static `DateDiff` method of the `DBFunction` class to compute the interval between two `java.util.Date` fields. The `DateDiff` method takes two `ColumnRef` parameters to timestamp fields in the database: a starting timestamp and an ending timestamp. The `DateDiff` method returns the interval between the two fields.

```
DateDiff(dateDiffPart : DateDiffPart, startDate : ColumnRef, endDate : ColumnRef) : DBFunction
```

You use the initial parameter of the method, `dateDiffPart`, to 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 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
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

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(
        QuerySelectColumns.path(Paths.make(Activity#AssignmentDate))))) {
  print("Assigned on " + activity.AssignmentDate + ": " + activity.DisplayName)
}
```

Effect of daylight saving time on the return value of the DateDiff method

The `DateDiff` method does not adjust for daylight saving, or summer, time. For example, consider a locale in which daylight saving time ends on the first Sunday in November. A call to the `DateDiff` method requests the number of hours between the `java.util.Date` values `2017-11-04 12:00` and `2017-11-05 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 `java.util.Date` 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. The `DatePart` method returns the extracted part as a `java.lang.Number`.

```
DatePart(datePart : DatePart, date : ColumnRef) : DBFunction
```

For the `datePart` parameter, you can specify `HOUR`, `MINUTE`, `SECOND`, `DAY_OF_WEEK`, `DAY_OF_MONTH`, `MONTH`, or `YEAR`. When you specify `DAY_OF_WEEK` as the part to extract, the first day of the week is Monday.

The following 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
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

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(
        QuerySelectColumns.path(Paths.make(Activity#AssignmentDate))))) {
  print("Assigned on " + activity.AssignmentDate + ": " + activity.DisplayName)
}
```

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 `java.util.Date` field to use in a comparison predicate. The `DateFromTimestamp` method takes a single parameter, a column reference to a `java.util.Date` field. The return value is a `java.util.Date` with only the date portion specified.

```
DateFromTimestamp(timestamp : ColumnRef) : DBFunction
```

The following Gosu code uses the `DateFromTimestamp` method to extract the date from the creation 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
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths
uses gw.api.util.DateUtil

// Make a query of Address instances.
var query = Query.make(Address)
```

```
// Query for addresses created today.
query.compare(DBFunction.DateFromTimestamp(query.getColumnRef("CreateTime")),
              Equals, DateUtil.currentDate())

// Order the result by creation date and iterate the items fetched.
for (address in query.select().orderBy(QuerySelectColumns.path(Paths.make(Address#CreateTime)))) {
  print(address.DisplayName + ": " + address.CreateTime)
}
```

Comparison of a date and time field to a range of values

In SQL, you use the `BETWEEN` keyword to apply comparison predicates to an inclusive range of column values in the `WHERE` clause. With the query builder APIs, you use the `between` method to apply a range predicate to entity fields. The following Gosu code applies the `between` method to values in the creation timestamp field of `Address` entity instances to select addresses that were created in the previous month.

```
uses gw.api.database.Query
uses gw.api.database.DBFunction
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths
uses gw.api.util.DateUtil

// Query the Company instances for a range of creation dates.
var firstDayOfCurrentMonth = DateUtil.currentDate().FirstDayOfMonth
var previousMonthStart = DateUtil.addMonths(firstDayOfCurrentMonth, -1)
var previousMonthEnd = DateUtil.addDays(firstDayOfCurrentMonth, -1)

// Make a query of Address instances.
var query = Query.make(Address)
// Query for addresses created in the previous month.
query.between(DBFunction.DateFromTimestamp(query.getColumnRef("CreateTime")),
              previousMonthStart, previousMonthEnd)

// Order the result by creation date and iterate the items fetched.
for (address in query.select().orderBy(QuerySelectColumns.path(Paths.make(Address#CreateTime)))) {
  print(address.DisplayName + ": " + address.CreateTime)
}
```

See also

- “Handling of null values in a range comparison” on page 325

Using a comparison predicate with a null value

In a relational database, you can define columns that allow null values or that require every row to have a value. The equivalent in a PolicyCenter application is to 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 code returns all `Person` instances where the birthday is unknown.

```
uses gw.api.database.Query

var query = Query.make(Person)
query.compare(Person#DateOfBirth, Equals, null)
```

The following Gosu code returns all `Address` instances where the first address line is known.

```
uses gw.api.database.Query

var query = Query.make(Address)
query.compare(Address#AddressLine1, NotEquals, null)
```

Handling of null values in a range comparison

If one of the two comparison values that you pass to the `between` method is `null`, Gosu performs a simple comparison that is equivalent to calling the `compare` method. If the first comparison value is `null`, Gosu performs a less-than-or-equal comparison on the second value. If the second comparison value is `null`, Gosu performs a greater-than-or-equal comparison on the first value. For example, the following two statements are equivalent:

```
query.between(Company#Name, null, "Business")
query.compare(Company#Name, LessThanOrEquals, "Business")
```

How null values get in the database

Null values get in the database only for entity properties that the *Data Dictionary* does not define as `non-null`. To assign `null` values to entity instance properties, use the special Gosu value `null`. The following Gosu code sets an `int` property and a `java.util.Date` property to `null` on a new entity instance.

```
var aPerson = new Person()

aPerson.DateOfBirth = null      // Set a java.util.Date to null in the datbase
aPerson.NumDependents = null    // Set an int to null in the database
```

After the bundle with the new `Person` instance commits, its `DateOfBirth` and `NumDependents` properties are `null` in the database.

Blank and empty strings become null in the database

To assign `null` values to `String` properties, use the special Gosu value `null` or the empty string (`""`). If you set the property of an entity instance to a blank or empty string, PolicyCenter coerces the value to `null` when it commits the instance to the database.

The following Gosu code sets three `String` properties to different values.

```
var anAddress = new Address()

anAddress.AddressLine1 = " "    // Sets a String to null in the database
anAddress.AddressLine2 = ""     // Sets a String to null in the database
anAddress.AddressLine3 = null   // Sets a String to null in the database
```

After the bundle with the new `Address` instance commits, all three address lines are `null` in the database. Before PolicyCenter commits `String` values to the database, it trims leading and trailing spaces. If the result is the empty string, PolicyCenter 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.

Controlling whether PolicyCenter trims whitespace before committing string properties

You can control whether PolicyCenter 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 PolicyCenter from trimming whitespace before committing a `String` property to the database, add the `trimwhitespace` column parameter in the column definition, and set the parameter to `false`. The XML text of a column definition that does not trim whitespace looks like the following:

```
<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 PolicyCenter trims leading and trailing spaces. You cannot configure whether PolicyCenter coerces an empty string to null.

See also

- "String variables can have content, be empty, or be null" on page 59

Using set inclusion and exclusion predicates

In SQL, you can embed an SQL query in-line within another query. A query that is embedded inside another query is called a *subselect* or a *subquery*. The SQL query language does not provide a keyword to create a subselect. The structure of your SQL query creates a subselect, as shown in the following SQL syntax, which 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%');
```

The following example uses a subselect to find users that have created notes of the specified topic types:

```
SELECT ID FROM pc_user
  WHERE ID
    IN (SELECT AuthorID FROM pc_note
          WHERE Topic IN (1, 10006));
```

The following Gosu code constructs and executes a functionally equivalent query to the preceding SQL example.

```
uses gw.api.database.Query
uses gw.api.database.InOperation

var outerQuery = Query.make(User) // Returns User Query
var innerQuery = Query.make(Note) // Returns Note Query

// Filter the inner query
innerQuery.compareIn(Note#Topic, {NoteTopicType.TC_GENERAL, NoteTopicType.TC_LEGAL})
// Filter the outer query by using a subselect
outerQuery.subselect(User#ID, InOperation.CompareIn, innerQuery, Note#Author )

// Alternatively, use simpler syntax if you do not need to filter the inner query
// queryParent.subselect(User#ID, InOperation.CompareIn, Note#Author )

var result = outerQuery.select() // Returns User query Object result
for (user in result) {           // Execute the query and iterate the results
  print(user.DisplayName)
}
```

The `subselect` method is overloaded with many method signatures for different purposes. The preceding Gosu code shows two of these 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. These methods do not accept a `null` value in the list of values. A run-time error occurs if you provide a `null` value in set of comparison values.

Comparing column values with each other

Some predicate methods have signatures that let you compare a column value on an entity instance with another column value. These methods use a property reference to access the first column and a column reference to access the second column. Use the `getColumnRef` method on query, table, and restriction objects to obtain a column reference.

Note: The `ColumnRef` object that the `getColumnRef` method returns does not implement the `IQueryBuilder` interface. You cannot chain the `getColumnRef` method with other query builder methods.

For example, you need 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 pc_contact
  WHERE EmailAddress1 <> EmailAddress2;
```

The following Gosu code constructs and executes a functionally equivalent query to the preceding SQL statement. This Gosu code uses the `getColumnRef` method to retrieve the second email address property.

```
uses gw.api.database.Query

var query = Query.make(Contact) // Query for contacts where email 1 and email 2 do not match.
query.compare(Contact#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)
}
```

See also

- "Chaining query builder methods" on page 375

Comparing a column value with a literal value

You can pass a Gosu literal as a comparison value to predicate methods. PolicyCenter generates and sends SQL that treats the literal value as an SQL query parameter to the relational database. Use the `DBFunction.Constant` static method to specify literal values to treat as SQL literals in the SQL query that PolicyCenter 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. Using this method to coerce a Gosu literal to an SQL literal can improve query performance. To improve query performance, you must 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 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(Address#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 Gosu 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(Address#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

The `DBFunction.Constant` method has these signatures:

```
Constant(value String) : DBFunction
Constant(value java.lang.Number) : DBFunction
Constant(dataType IDataType, value Object) : DBFunction
```

Comparing a typekey column value with a typekey literal

Often you want to select data based on the values of typekey fields. A *typekey field* takes its values from a specific typelist. A *typelist* contains a set of codes and related display values that appear in the drop-down lists of the PolicyCenter application.

Gosu provides *typekey literals* with which 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 code specifies the typekey code for `open` in 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(Address#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 through the typelist names to the typekey code that you need.

See also

- "Typekeys and typelists" on page 64
- *Configuration Guide*

Using a comparison predicate with spatial data

Often, you want to compare one geographical location to another geographical location with a distance calculation. In particular, you desire to select entity instances having a property that pinpoints a location that is within a given distance of a reference location. To do this, use the appropriate `withinDistance` method in connection with an entity query.

The following code examples demonstrate how to use two varieties of the overloaded `withinDistance` method. The first code example uses the `withinDistance` method in connection with a simple entity query. This query selects those `Address` entity instances with a `SpatialPoint` property that is within 10 miles of a `SpatialPoint` constant representing San Mateo. The first code example is as follows:

```
uses gw.api.database.spatial.SpatialPoint
uses gw.api.database.Query

// Make a query on the Address entity.
var addressQuery = Query.make(Address)

// Define the location of the center point for the distance calculation.
var SAN_MATEO = new SpatialPoint(-122.300, 37.550)

// Specify the spatial point in the address instance and the maximum distance from the center.
addressQuery.withinDistance(Address.SPATIALPOINT_PROP.get(), SAN_MATEO, 10, UnitOfDistance.TC_MILE)

// Indicate that the query is an entity query by selecting all columns.
var result = addressQuery.select()

// Print the results of the query.
for (address in result) {
  print(address)
}
```

The second code example uses the `withinDistance` method in connection with a more complex entity query that joins a primary entity type with a related entity type. This query joins the `Person` entity type, which has a `PrimaryAddress` property of type `Address`, with the `Address` entity type. The query then selects those `Person` entity instances with a `PrimaryAddress.SpatialPoint` property that is within 10 miles of a `SpatialPoint` constant representing San Mateo. The second code example is as follows:

```
uses gw.api.database.spatial.SpatialPoint
uses gw.api.database.Query

// Make a query on the Person entity.
var personQuery = Query.make(Person)

// Explicitly join the Address entity for the primary address to the Person.
var addressTable = personQuery.join(Person#PrimaryAddress)

// Define the location of the center point for the distance calculation.
var SAN_MATEO = new SpatialPoint(-122.300, 37.550)

// Specify the spatial point in the person's primary address and the maximum distance from the center.
addressTable.withinDistance(Address.SPATIALPOINT_PROP.get(), "Person.PrimaryAddress.SpatialPoint", SAN_MATEO, 10,
UnitOfDistance.TC_MILE)

// Indicate that the query is an entity query by selecting all columns.
var result = personQuery.select()

// Print the results of the query.
for (person in result) {
  print(person + ", " + person.PrimaryAddress)
}
```

Note: You cannot display the distance between the `SpatialPoint` property of an entity instance and a `SpatialPoint` constant in an entity query. To do so would require a call to the `Distance` static method in the `DBFunction` class, which is incompatible with entity queries. Only a row query can use a static method in the `DBFunction` class to return a value. To display distances in conducting a comparison, see "Using a distance function as a column selection" on page 351.

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 record that you need, both expressions must be true. You need a person's last name to be "Smith" and that same person's first name to be "John." You do not want only one of the expressions to be true. You do not want to select people whose last name is "Smith" or first name is "John."

For other requirements, you need to combine predicate expressions, any of which may 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 need, either expression may be true. You want an address to have either "Chicago" or "Los Angeles" as the city.

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(Contact#LastName, Equals, "Smith")
query.compare(Contact#FirstName, Equals, "John")
```

You can also use the `and` method to make the `AND` combination of the predicates explicit. For example, the following code is equivalent to the previous example:

```
query.and( \ andCriteria -> {
  andCriteria.compare(Contact#LastName, Equals, "Smith")
  andCriteria.compare(Contact#FirstName, Equals, "John")
})
```

See also

- "Using Boolean algebra to combine sets of predicates" on page 331

Using OR to combine predicates that require at least one to be true

In SQL, you combine predicates with the `OR` keyword if one or more 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 one or more must be true.

```
query.or( \ orCriteria -> {
  orCriteria.compare(Address#City, Equals, "Chicago")
  orCriteria.compare(Address#City, Equals, "Los Angeles")
})
```

Using Boolean algebra to combine sets of predicates

You can use Boolean algebra to combine predicates with logical AND and OR. You use the and and or methods on queries, tables, and restrictions. The and and or methods modify the SQL query and add a clause to the query. To combine predicates, you use Gosu blocks, which are functions that you define in-line within another function.

Predicate linking mode

The default behavior of a new query is to use an implicit logical AND between its predicates. This behavior is known as a *predicate linking mode* of AND. The way that you apply and and or methods to a query can change the linking mode of a new series of predicates.

The and and or methods change only the linking mode of the predicates that are defined in the block that you pass to the method. The and and or methods do not change the linking mode of existing predicates. For example, if you apply multiple or methods directly to a query, the OR predicates are joined by an AND.

The or method has the following effect on the predicate linking mode of a query:

- To existing restrictions, add one parenthetical phrase.
- Link the new parenthetical phrase to previous restrictions in the current linking mode.
- The block passed to the or method includes a series of predicates. Use a predicate linking mode of OR between predicates inside the block.
- A predicates defined in the block could be another AND or OR grouping, containing additional predicates with another usage of the and or or methods.

The and method has the following effect on the predicate linking mode of a query:

- To existing restrictions, add one parenthetical phrase.
- Link the new parenthetical phrase to previous restrictions in the current linking mode.
- The block passed to the and method includes a series of predicates. Use a predicate linking mode of AND between predicates inside the block.
- A predicate defined in the block could be another AND or OR grouping, containing additional predicates with another usage of the and or or methods.

Syntax of Boolean algebra methods for predicates

The syntax of the or method is:

```
query.or( \ OR_GROUPING_VAR -> {
  OR_GROUPING_VAR.PredicateOrBooleanGrouping(...)
  OR_GROUPING_VAR.PredicateOrBooleanGrouping(...)
  [...]
}
```

The syntax of the and method is:

```
query.and( \ AND_GROUPING_VAR -> {
  AND_GROUPING_VAR.PredicateOrBooleanGrouping(...)
  AND_GROUPING_VAR.PredicateOrBooleanGrouping(...)
  [...]
}
```

Each use of *PredicateOrBooleanGrouping* could be either:

- A predicate method such as compare or between.
- Another Boolean grouping by calling the or or and method.

Within the block, you call the predicate and Boolean grouping methods on the argument that you pass to the block. Do not call the predicate and Boolean grouping methods on the original query.

In the syntax specifications, *OR_GROUPING_VAR* and *AND_GROUPING_VAR* refer to a grouping variable name that identifies the OR or AND link mode in each peer group of predicates. Guidewire recommends using a name for the block parameter variable that indicates one of the following:

- **Name describes the linking mode** – Use a variable name that specifies the linking mode between predicates in that group, such as or1 or and1. You cannot call the variable or or and because those words are language keywords. Add a digit or other unique identifier to the words or or and.
- **Name with specific semantic meaning** – For example, use carColors for a section that checks for car colors.

Combining AND and OR groupings

You can combine and populate 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, the following alternative approaches might provide better performance in production, depending on your data:

- If the OR clauses all test a single property and all of the values are non-null, rewrite and collapse the tests into a single comparison predicate using the compareIn method, which takes a list of non-null values.

  For example, if your query checks a property for a color value for matching any one of 30 color values, create a Collection that contains the color values. You can use any class that implements Collection, such as an array list or a set. Use the Collection as an argument to compareIn:

```
var lastNamesArrayList = {"Smith", "Applegate"}
var query = Query.make(Person)
query.compareIn(Person.LASTNAME_PROP.get(), lastNamesArrayList)

// Fetch the data with a for loop.
var result = query.select()
for (person in result) {
  print (person)
}
```

- 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

Typical predicate definitions use a pattern that uses one line of code for each predicate in the block. Using or in the name of the Gosu block variable more closely matches an English construction that describes the final output because an OR clause separates the predicates. Consider using the pattern in the following example, and avoid chaining the predicates together.

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

query.or( \ or1 -> {
  or1.compare(Person#CreateTime, GreaterThanOrEquals,
          DateUtil.addBusinessDays(DateUtil.currentDate(), -10))
  or1.compare(Person#LastName, Equals, "Newton")
  or1.compare(Person#FirstName, Equals, "Ray")
})
```

In theory, you could use method chaining to use a single line of code, rather than using three separate lines. In this case, chaining might make the code harder to understand.

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

query.or( \ or1 -> {
  or1.compare(Person#CreateTime, GreaterThanOrEquals, DateUtil.addBusinessDays(DateUtil.currentDate(),
    -10)).compare(Person#LastName, Equals, "Newton").compare(Person#FirstName, Equals, "Ray")
})
```

Examples

For example, the following Gosu code links three predicates together with logical `OR`. The query returns rows if any of the three predicates are true for that row.

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

// Find rows with creation date in the last 10 business days, last name of Newton, or first name of Ray
query.or( \ or1 -> {
  or1.compare(Person#CreateTime, GreaterThanOrEquals,
          DateUtil.addBusinessDays(DateUtil.currentDate(), -10))
  or1.compare(Person#LastName, Equals, "Newton")
  or1.compare(Person#FirstName, Equals, "Ray")
})
```

For example, the following Gosu code links three predicates together with logical `AND`. The query returns rows if all three predicates are true for that row.

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

// Find rows with last name of Newton, first name of Ray, and creation date more than 14 days ago
query.and( \ and1 -> {
  and1.compare(Person#LastName, Equals, "Newton")
  and1.compare(Person#FirstName, Equals, "Ray")
  and1.compare(Person#CreateTime, LessThanOrEquals, DateUtil.addDays(DateUtil.currentDate(), -14))
})
```

This code is functionally equivalent to simple linking of predicates, because the default linking mode is `AND`. The following Gosu code shows the use of predicate linking to achieve the same query.

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

query.compare(Person#LastName, Equals, "Newton")
query.compare(Person#FirstName, Equals, "Ray")
query.compare(Person#CreateTime, LessThanOrEquals, DateUtil.addDays(DateUtil.currentDate(), -14))
```

The power of the query building system is the ability to combine `AND` and `OR` groupings. For example, suppose we need to represent the following pseudo-code query predicates:

```
(CreateTime < 10 business days ago) OR ( (LastName = "Newton") AND (FirstName = "Ray") )
```

The following code represents this pseudo-code query by using Gosu query builders:

```
query.or( \ or1 -> {
  or1.compare(Person#CreateTime, LessThanOrEquals,
          DateUtil.addBusinessDays(DateUtil.currentDate(), -10))
  or1.and(\ and1 -> {
  and1.compare(Person#LastName, Equals, "Newton")
  and1.compare(Person#FirstName, Equals, "Ray")
  })
})
```

The outer `OR` contains two items, and the second item is an `AND` grouping. This structure directly matches the structure of the parentheses in the pseudo-code.

Similarly, suppose we need to represent the following pseudo-code query predicates:

```
( (WrittenDate < Today - 90 days) OR NOT ISNULL(CancellationDate) )
AND
( (BaseState = "FR") OR Locked )
```

The following code represents this pseudo-code using Gosu query builders:

```
query.or( \ or1 -> {
  or1.compare(DBFunction.DateFromTimestamp(or1.getColumnRef("WrittenDate")), LessThan,
```

```
              DateUtil.addDays(DateUtil.currentDate(), -90))
  or1.compare(PolicyPeriod#CancellationDate, NotEquals, Null)
})
query.or( \ or2 -> {
  or2.compare(PolicyPeriod#BaseState, Equals, Jurisdiction.TC_FR)
  or2.compare(PolicyPeriod#Locked, Equals, true)
})
```

Each `OR` contains two items, and the two `OR` items are combined using the default predicate linking mode of `AND`. This structure directly matches the structure of the parentheses in the pseudo-code.

See also

- "Blocks" on page 175

Joining a related entity to a query

To build useful queries in PolicyCenter, you often must join related entities to the primary entity of the query. Typical reasons to create a join are to restrict the set of rows that the query returns or to retrieve additional information that the related entities contain. The query builder APIs support setting restrictions only on database-backed fields, which store their values directly in the database. Primary entities have many properties that are arrays, foreign keys, type lists, and derived fields. Related tables provide the database-backed fields for these types of properties. 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.

Joining an entity to a query with a simple join

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

Joining tables in SQL SELECT statements

In SQL, you can join two tables based on columns with matching values to form a new, virtual *join table*. The rows in the join table contain columns from each original table. On the join table, you can specify restrictions, which can have columns from both tables. When the SQL query runs, the result set has rows and columns from the join table. A join typically involves a column in one table that contains a foreign key value that matches a primary key value in a column in a second table. A *primary key* is a column with values that uniquely identify each row in a database table. A *foreign key* is a column in one table that contains values of a primary key in another table. The definition of a foreign key column specifies whether the values in the column are unique.

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 addresses.ID = companies.primary_address;
```

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

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	

companies:

id	name	primary_address
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

The columns in the result set are a union of the columns in the `companies` and `addresses` tables. 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.

Joining entities with the query builder API

With the query builder API, 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 API construct an internal object to represent the joined entity and its participation in the query.

In the simplest case, use the `join` method to join a dependent 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 dependent entity. Use the Gosu property name, not the actual column name in the database.

For example, the following Gosu code shows how to join the `Address` entity to a query of the `Company` entity. The code uses the `PrimaryAddress` property of `Company`, which is a foreign key to `Address`.

```
uses gw.api.database.Query

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

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

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.

A significant difference from SQL is that the query builder API return only instances of the primary entity type. With the query builder API, you 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(Company#PrimaryAddress) // Join Address entity to the query by a foreign key on Company
var select = query.select()

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

The preceding Gosu code fetches the `Address` instance lazily from the database. If the record is not available in the cache, this code could cause the execution of a query for every company. To reduce the number of database queries, you can use a row query to retrieve the complete set of rows and the necessary columns from the primary and joined tables.

See also

- “Working with row queries” on page 347

Ways to join a related entity to a query

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 outer full joins are not supported.

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 provides the foreign key nor whether the foreign key is defined in metadata. 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.

The query builder API uses different signatures on the `join` method depending on which side of a join provides the foreign key.

- `join(primaryEntity#column)` – Joins two entities that have the foreign key is on left
- `join(secondaryEntity#column)` – Joins two entities that have the foreign key is on the right.
- `join("columnPrimary", table, "columnDependent")` – Joins two entities without regard to foreign keys.

Making an inner join with the foreign key on the left

In SQL, an inner join is the default type of join. You make an inner join with the `JOIN` keyword. To be explicit about the type of join, use `INNER JOIN`. You specify the table that you need to join to the primary table of the query. You use the `ON` keyword to specify the columns that join the tables. By convention, you specify the join column on the primary table first. In the following SQL statement, the foreign key, `PrimaryAddressID`, is on the primary table, `pc_contact`. Because SQL considers the primary table to be on the left side, the foreign key is on the left side of the join.

```
SELECT * FROM pc_contact
  INNER JOIN pc_address
    ON pc_address.ID = pc_contact.PrimaryAddressID;
```

With the query builder API, you use the `join` method to specify an inner join. Use a property reference of the primary entity and property name as the 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 dependent entity. The query builder API uses metadata to determine the entity type to which the foreign key relates.

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

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

Unlike SQL, you do not specify which dependent entity to join, in this case `Address`. Neither do you specify the property on the dependent entity, `ID`. The query builder API uses metadata from the *Data Dictionary* to provide the missing information.

Making an inner join with the foreign key on the right

In SQL, an inner join is the default type of join. You make an inner join with the `JOIN` keyword. To be explicit about the type of join, use `INNER JOIN`. You specify the table that you need to join to the primary table of the query. You use the `ON` keyword to specify the columns that join the tables. By convention, you specify the join column on the primary table first. In the following SQL statement, the foreign key, `UpdateUserID`, is on the secondary table, `pc_address`. Because SQL considers the primary table to be on the left side, the foreign key is on the right side of the join. The query returns a result with all addresses and users who lasted updated them. The result contains all columns in the address and user tables.

```
SELECT * FROM pc_user
  INNER JOIN pc_address
    ON pc_user.ID = pc_address.UpdateUserID;
```

With the query builder APIs, you use the `join` method to specify an inner join. Use a property reference of the dependent entity and property name as a single parameter if the dependent entity has the foreign key. In these cases, the foreign key is on the right. The database must have a column that contains the foreign key. The data model definition of the column must specify `foreignkey`. You cannot use a column that is a `onetoone` or `edgeForeignKey` type because these types do not use a database column. To confirm that a foreign key uses a database field, in the *Data Dictionary*, check that the foreign key does not have “(virtual property)” text after its name.

Joining to a dependent entity with the foreign key on the right

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

```
var queryUser = Query.make(User)
queryUser.join(Address#UpdateUser)
```

Similarly to SQL, you must specify the dependent entity to join, which in this case is `Address`. Unlike SQL, you do not specify the property on the primary entity, `ID`. The query builder APIs use metadata from the Data Dictionary to provide the missing information.

Applying predicates to the dependent entity

You seldom want to retrieve all instances of the primary type of a query. To select a subset, you often join a dependent entity and apply predicates to one or more of its properties. You apply these predicates for a secondary table with the foreign key on the right in the same way as for predicates where the foreign key is on the left.

For example, you want to select only users who have updated addresses in the city of Chicago. You must join the `Address` entity to your `User` query. Then, you can apply the predicate `City = "Chicago"` to `Address`, the dependent entity. The SQL for this query looks like the following lines.

```
SELECT DISTINCT pc_user.PublicID FROM pc_user
  INNER JOIN pc_address
    ON pc_user.ID = pc_address.UpdateUserID
    AND pc_address.City = 'Chicago';
```

The following Gosu code represents the same SQL query.

```
uses gw.api.database.Query
uses gw.api.database.Relop

// -- 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(Address#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)
}
```

See also

- “Restricting query results by using fields on joined entities” on page 343
- “Working with row queries” on page 347
- “Handling duplicates in joins with the foreign key on the right” on page 341

Making an inner join without regard to foreign keys

In SQL, you can make an inner join between two tables regardless of any formal SQL declaration of a foreign key relation between the tables. The columns on which a SQL Select statement joins two tables must have values that match, so the data types of both columns must be the same. In addition, application logic, not a declared foreign key constraint, must ensure that both columns contain values that potentially match.

To use the query builder API to create an inner join that does not use a foreign key, the database must have columns that contain each join property. The data model definition of the column must specify `foreignkey` or `column`. You cannot use a virtual column, nor a column that is an `array`, `onetoone`, or `edgeForeignKey` type because these types do not use a database column. To confirm that a foreign key uses a database field, in the *Data Dictionary*, check that the foreign key does not have “(virtual property)” text after its name.

Test the use of an inner join that you make without regard to foreign keys in a realistic environment. Using this technique on a query can cause a full-table scan in the database if the query cannot use an index.

WARNING For a query on large tables, using an inner join without regard to foreign keys can cause an unacceptable delay to the user interface.

Comparing SQL syntax to query builder syntax

For an inner join in SQL, the `ON` clause specifies the columns that join the tables. The clause requires the names of both tables and both columns. The following SQL statement joins rows from the `addresses` table to rows in the `companies` table where values in `ID` match values in `primary_address`.

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

With the query builder API, you use the `join` method with the following signature to make inner joins of entities without regard to foreign keys in the data model.

```
join("columnPrimary", table, "columnDependent")
```

The first parameter is the name of a column on the primary entity of the query. The second and third parameters are the names of the dependent entity and the column that has matching values.

Example

In the following Gosu code, the primary entity `Note` has no declared foreign key relation with the dependent entity `Contact`. An indirect relation exists through their mutual foreign key relations with the `User` entity. Both entities in the query have a column that contains the user ID, so the join method can join the `Contact` entity to the query of the `Note` entity.

```
uses gw.api.database.Query

// -- query the Note entity --
var queryNote = Query.make(Note).withDistinct(true)

// -- select only notes where the same user created a contact.
queryNote.join("CreateUser", Contact, "CreateUser")

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

for (note in result) {
  print ("The user who created the note '" + note.Subject + "' also created a contact")
}
```

The `Note` entity has no direct relevant foreign key relation to `Contact`. You cannot use Gosu dot notation to navigate from notes in the result to properties on the contact.

Making a query with a left outer join

For 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.

Making a left outer join with the foreign key on the left

In SQL, you make a left outer join with the `LEFT OUTER 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 OUTER 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 dependent entity. The query builder APIs use metadata to determine the entity type to which the foreign key the relates.

Joining to a dependent entity with the foreign key on the left

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

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

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

Accessing properties of the dependent entity

Unlike SQL, the result of an entity query contains instances of the primary entity, not composite table rows with columns. Explicitly specifying a join by using the join or outerJoin methods does not change how you access properties of the dependent entity. Information from joined entities generally is not included in results. If the foreign key is on the left, you can access properties from the dependent entity by using dot notation. However, if the foreign key is on the right, you cannot use dot notation to traverse from primary instances on the left to related instances on the right.

Dependent entity with the foreign key on the left

The following Gosu code has the foreign key on the left. The primary entity type, `Company`, has a foreign key, `PrimaryAddress`, which relates to the dependent entity type, `Address`. You can use dot notation to access the properties of `Address`.

```
uses gw.api.database.Query

var queryCompany = Query.make(Company)
queryCompany.join(Company#PrimaryAddress)

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

Dependent entity with the foreign key on the right

The following Gosu code has the foreign key on the right. You can determine that a user updated addresses, but not which addresses were updated. You cannot use dot notation to retrieve information from the Address entity rows. This limitation does not affect processing the results of inner queries with foreign keys on the right if you need no information from the related instances that produced the result. If you do want to access the information, for example to display on a screen or in a report, you must use a row query.

```
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)

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

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

Handling duplicates in joins with the foreign key on the right

Duplicate occurrences of records from the primary table in a result can occur only for joins that have the foreign key on the right. If the foreign key for a join is on the left, the value for any primary instance is unique in the dependent table. A foreign key on the left relates either to:

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

A query that you create by using `join` or `outerJoin` with the foreign key on the right of the join often fetches duplicate instances of the primary entity. For example, the following Gosu code finds users that have updated addresses.

```
var queryParent = Query.make(User)
var tableChild = queryParent.join(Address#UpdateUser)
```

For the previous example, if multiple `Address` instances relate to the same `User`, the result includes duplicates of that `User` instance. One instance exists in the result of the join query for each child object of type `Address` that relates to that instance.

Duplicate instances of the primary entity in a query result are 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 set of rows is what you want. In cases for which you need to return a single row from the primary table, you must design and test your query to ensure this result.

For joins with the foreign key on the right, try to reduce duplicates on the primary table. The best way to reduce duplicates is to ensure that the secondary table only has one row that matches the entity on the primary table.

If you cannot eliminate duplicates in this way, other approaches can limit duplicates created because of a join:

- Rewrite to use the `subselect` method approach. For joins with the key on the right, the query optimizer often performs better with `subselect` than with `join` or `outerJoin`. However, using the `join` method might perform better in some cases. For example, consider using `join` if the dependent entity includes predicates that are not very selective and return many results. 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.

  ```
  var queryParent = Query.make(User)
  queryParent.subselect(User#ID, InOperation.CompareIn, Note#Author)
  ```

- 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 results to contain only a single row for each row of the parent table. To turn off this behavior later, call the method again and pass `false` as an argument. For example:

  ```
  var queryParent = Query.make(User)
  queryParent.withDistinct(true)
  queryParent.join(Note#Author)
  ```

- Add predicates to the secondary table to limit what your query matches. For example, only match entities on the primary table if a related child object has a certain property with a specific value or range of values.

The following example adds predicates after the join using properties on the joined table:

```
var queryParent = Query.make(User)
var tableChild = queryParent.join(Note#Author)
tableChild.compare(Note#CreateTime, GreaterThanOrEquals,
                   DateUtil.addDays(DateUtil.currentDate(), -5))
```

See also

- "Working with row queries" on page 347

Joining to a subtype of an entity type

Many PolicyCenter entity types have subtypes. The parent entity type and its subtypes all share the same database table. For example, the `Contact` entity type, its direct subtypes, `Company`, `Person`, and `Place`, and its indirect subtypes all have rows in the `pc_contact` database table. If you create a query that accesses a subtype as the primary entity, the query builder API adds appropriate filters to the query to return only rows of that subtype. You use the same syntax to access properties on the parent type and properties that are specific to the subtype. If you create a query that joins a subtype as a dependent entity, the query builder API joins the least restrictive entity type to the query.

For example, the following lines join organizations to contacts:

```
var queryOrg = Query.make(Organization)
var tableContact = queryOrg.outerJoin(Organization#Contact)
```

Because the `Contact` property exists on the `Organization` entity type, the following code also joins organizations to contacts and does not join only company records:

```
var queryOrg = Query.make(Organization)
var tableContact = queryOrg.outerJoin(Company#Contact)
```

If you want to access company properties or only need to see organizations that have a company as a contact, you must cast the joined table to `Company`. Use code like the following lines:

```
var queryOrg = Query.make(Organization)
var tableContact = queryOrg.join(Organization#Contact).cast(Company)
```

If the column that you use to join the dependent table is specific to a particular subtype of an entity type, the query builder API performs the more restrictive join. In this case, you do not need to cast the type of the joined table.

Restricting a result set by joining to a subtype of an entity type

You can restrict the result set of a query by requiring a specific subtype for the joined entity. You use the entity name and a property to specify the join. The query builder API generates an SQL query to join to the database table. If the property for the join exists on the subtype entity and not on the more general entity type, the generated SQL query adds a restriction for the subtype. If the property for the join exists on the more general entity type, the generated SQL query does not add a restriction for the subtype. To restrict the rows in the result set to those matching the subtype of the joined table, you must use the `cast` method.

For example, the following code prints the public IDs of policies that have ever been renewed:

```
uses gw.api.database.Query

var queryPolicy = Query.make(Policy).withDistinct(true)

// Join to any type of Job
// queryPolicy.join(Renewal#Policy)         // This join does not select only Renewal type jobs

// Join to only Renewal jobs
queryPolicy.join(Job#Policy).cast(Renewal)  // This join does select only Renewal type jobs
var result = queryPolicy.select()
```

```
for (policy in result) {
  print(policy.PublicID)
}
```

Accessing properties of a joined subtype of an entity type

If you need to access the properties of a specific subtype of an entity type, you must ensure that the query returns only rows for that subtype. If the subtype is the primary entity, you make a query by specifying the subtype name. If the subtype is a joined entity and the join uses a property on the parent entity type, you must cast the table to the required subtype. Additionally, if you need to access a property on the joined entity subtype, you must cast the property on the primary entity to the specific subtype.

For example, the following code prints information about activities for submission jobs and submission information about those jobs.

```
uses gw.api.database.Query

var queryActivity = Query.make(Activity)

// Join to only Submission jobs
var tableJob = queryActivity.join(Activity#Job).cast(Submission)

var result = queryActivity.select()

for (a in result) {
  // Use a variable to access properties on the Submission entity type
  var sj = a.Job as Submission
  print(a + " " + a.UpdateUser + " " + a.PolicyPeriod.PolicyNumber + " " + sj.BindOption)
}
```

Restricting query results by using fields on joined entities

You seldom want to retrieve all instances of the primary type of a query. In many cases, you join a dependent entity to a query so you can restrict the results by applying predicates to one or more of its properties. To apply predicates to the dependent 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 dependent instances to include in the result. That restriction also restricts the primary entity instances that the result returns.

Applying predicates to the dependent entity of an inner join

For example, you want a query that returns all companies in the city of Chicago. The SQL statement for this query looks like:

```
SELECT * FROM pc_contact
  JOIN pc_address
    ON pc_address.ID = pc_contact.PrimaryAddressID
  WHERE pc_contact.Subtype IN (1,8)
    AND pc_address.City = 'Chicago';
```

Using the query API, 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. You need to join the `Address` entity to your query. You must capture the object reference that the `join` method returns so that you can specify predicates on values in `Address`.

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

Alternatively, you can chain the `join` method and call the predicate method on the returned Table object in a single statement, as the following Gosu code shows. If you use this pattern, you do not need to make a variable for the Table object.

```
queryCompany.join(Company#PrimaryAddress).compare(Address#City, Equals, "Chicago")
```

When you run the query, the result contains companies that have Chicago as the city on their primary addresses. The following code demonstrates the use of this predicate on a joined entity.

```
uses gw.api.database.Query
uses gw.api.database.Relop

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

// Join Address as the dependent entity to the pimary entity Company.
var tableAddress = queryCompany.join(Company#PrimaryAddress)

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

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

Running this code produces a list like the following one:

```
Jones and West Insurance, Chicago
Shumway Contracting, Chicago
New Energy Corp., Chicago
...
Auto Claims of the West, Chicago
Auto Claims Defenders, Chicago
```

The query result of the previous example contains only `Company` instances. Even though the code includes the `Address` entity in the query, the results do not include information from joined entities. The code example 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`.

Predicates on the left outer join dependent entity differ from SQL

With the query builder APIs, generally you join a dependent entity to a query only so you can restrict the results with predicates on the dependent entity. Using the compare method to apply a predicate to a dependent entity that you join with a left outer join has no effect on the rows that Gosu retrieves.

The ways that SQL queries and Gosu code filter the dependent table rows differ. Gosu fetches the dependent entity only if necessary, if code that accesses the dependent entity executes. Gosu applies the filter when running the query on the primary table, not when fetching the dependent table row. The value of a dependent entity property that you access by using dot notation is `null` only if there is no matching entity instance regardless of the filter. To retrieve filtered values from related entities, use a row query.

For example, the following SQL filters the names of group supervisors. If the name does not match, the value for the supervisor user and contact columns are `null`.

```
SELECT * FROM groups
  LEFT OUTER JOIN users
    ON groups.supervisor = users.ID
  LEFT OUTER JOIN contacts
    ON users.contactID = contacts.ID
    AND contacts.lastName = 'Visor';
```

The following Gosu code shows the effect of dot notation to access information related to the primary entity of an outer-join 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. The output shows the supervisor name for every group because Gosu applies the filter on the primary table query, not when `print` retrieves the property values for related entities.

```
uses gw.api.database.Query

// -- Query the Group entity --
var queryGroup = Query.make(Group)
// -- Supervisor is a foreign key from Group to User.
var userTable = queryGroup.outerJoin(Group#Supervisor)
// -- Contact is a foreign key from User to UserContact.
var contactTable = userTable.outerJoin(User#Contact)
// -- Apply a filter to the outer joined entity
contactTable.compare(UserContact#LastName, Relop.Equals, "Visor")
// -- Fetch the Group instances with a for loop and print them --
var result = queryGroup.select()

for (group in result) {
  if (group.Supervisor != null) {
    // Every supervisor's name appears because the outer join does not filter the groups
    print(group.Name + ": " + group.Supervisor.Contact.DisplayName)
  } else { // Provide user-friendly text if the group has no supervisor
    // This text appears only for groups that have no supervisor
    print (group.Name + ": " + "This group has no supervisor")
  }
}
```

See also

- "Restricting a query with predicates on fields" on page 317
- "Restricting query results with fields on primary and joined entities" on page 345
- "Working with row queries" on page 347

Restricting query results with fields on primary and joined entities

You can restrict the result set of a query by applying predicates to both the primary and joined entities. You use different techniques to add the predicates depending on whether you want both (`AND`) or either (`OR`) to apply.

Using AND to combine predicates on primary and joined entities

If you need both predicates to apply to the result set, you can use a predicate method on the primary query and another predicate method on the secondary table.

For example, you want a query that returns all companies having a name that matches "Stewart Media" in the city of Chicago. The SQL statement for this query looks like:

```
SELECT * FROM pc_contact
  JOIN pc_address
    ON pc_address.ID = pc_contact.PrimaryAddressID
  WHERE pc_contact.Subtype IN (1,8)
    AND pc_contact.Name = 'Stewart Media'
    AND pc_address.City = 'Chicago';
```

Using the query API, the primary entity type of your query is `Company`, because you want instances of that type in your result. You add the predicate that specifies the restriction on the `Name` property.

```
var queryCompany = Query.make(Company).compare(Company#Name, Equals, "Stewart Media")
```

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

```
var tableAddress = queryCompany.join(Company#PrimaryAddress)
tableAddress.compare(Address#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
uses gw.api.database.Relop

// Start a new query with Company as the primary entity and restrict the result by company name.
var queryCompany = Query.make(Company).compare(Company#Name, Equals, "Stewart Media")

// Join Address as the dependent entity to the pimary entity Company.
var tableAddress = queryCompany.join(Company#PrimaryAddress)

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

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

Running this code produces a list like the following one:

```
Stewart Media, Chicago
```

You can add predicates to the primary entity of a query before or after you join a new entity to the query. For example, the following code adds a predicate to the primary entity before joining the secondary entity:

```
uses gw.api.database.Query
uses gw.api.database.Relop

// -- Query the Contact entity --
var queryContact = Query.make(Contact)
// -- Select only contacts that have a primary phone number that is a work phone
queryContact.compare(Contact#PrimaryPhone, Equals, typekey.PrimaryPhoneType.TC_WORK)
// -- Select only Contact instances that have a primary address in the city of Chicago --
queryContact.join(Contact#PrimaryAddress).compare(Address#City, Equals, "Chicago")
```

The following code adds a predicate to the primary entity after joining the secondary entity, and is equivalent to the previous code example:

```
uses gw.api.database.Query
uses gw.api.database.Relop

// -- Query the Contact entity --
var queryContact = Query.make(Contact)

// -- Add a predicate on the secondary entity
// -- Select only Contact instances that have a primary address in the city of Chicago --
var tableAddress = queryContact.join(Contact#PrimaryAddress)
tableAddress.compare(Address#City, Equals, "Chicago")

// -- Add a predicate on the primary entity
// -- Select only contacts that have a primary phone number that is a work phone
queryContact.compare(Contact#PrimaryPhone, Equals, typekey.PrimaryPhoneType.TC_WORK)
```

Using OR to combine predicates on primary and joined entities

If you need either predicate to apply to the result set, you use an `or` method on the secondary table. To refer to a column on the secondary table in the `or` block, you can use a property reference. To refer to a column on the primary table in the `or` block, you must use a column reference. A property on the primary table is not available in the context of the secondary table.

For example, you want a query that returns all companies either having a name that matches “Armstrong Cleaners” or in the city of Chicago. The SQL statement for this query looks like:

```
SELECT * FROM pc_contact
  JOIN pc_address
    ON pc_address.ID = pc_contact.PrimaryAddressID
```

```
  WHERE pc_contact.Subtype IN (1,8)
    AND (pc_contact.Name = 'Armstrong Cleaners'
     OR pc_address.City = 'Chicago');
```

Using the query API, 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. You need to join the `Address` entity to your query. You must capture the object reference that the `join` method returns so that you can specify predicates on values in `Address`. You use an `or` block to add the predicate that specifies the restriction on the `Name` property in the `Company` entity. You must use a column reference to access the `Name` column because that column is not in the `Address` entity.

```
var tableAddress = queryCompany.join(Company#PrimaryAddress)
tableAddress.compare(Address#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
uses gw.api.database.Relop

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

// Join Address as the dependent entity to the pimary entity Company.
var tableAddress = queryCompany.join(Company#PrimaryAddress)

// Add both predicates on the dependent entity.
tableAddress.or( \ or1 -> {
  or1.compare(Address#City, Equals, "Chicago")
  or1.compare(queryCompany.getColumnRef("Name"), Equals, "Armstrong Cleaners")
})

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

Running this code produces a list like the following one:

```
Armstrong Cleaners, San Ramon
Jones and West Insurance, Chicago
Shumway Contracting, Chicago
New Energy Corp., Chicago
...
Auto Claims of the West, Chicago
Auto Claims Defenders, Chicago
```

See also

- “Restricting a query with predicates on fields” on page 317
- “Restricting query results by using fields on joined entities” on page 343

Working with row queries

Certain kinds of SQL results require row queries instead of entity queries. The results of entity queries are entity instances. The entire object graphs of entity instances are available on demand. The results of row queries are row objects that provide only the properties that you select.

Overview of row queries

Entity queries cannot produce certain kinds of SQL results. Entity queries have the following characteristics:

- They cannot produce the results that SQL aggregate functions provide.
- They cannot produce results that involve filtering the rows in the dependent table on a SQL outer join.
- They load the complete object row into the application cache for selected entity instances, even though you often need a only few columns of data from the database.
- They provide access to related entity instances by making additional queries to the database to access entity instances that are not in the cache. These queries also load entire rows from the database tables.

In many cases, a row query can provide more suitable results than an entity query. A row query uses a single database query on both the main entity and related entities to provide only the subset of properties that you need.

If you need to view the results of your query in a list view, you must convert a row query to another format. List views cannot interpret the results of a row query. Consider whether an entity query on a view entity can provide more straightforward access to a subset of columns on primary and related entities.

See also

- "Limitations of row queries" on page 355
- *Configuration Guide*

Setting up row queries

You set up row queries by passing column selection parameters to the `select` method on query objects. The columns that you select become the columns of data in the row query result. The columns that you select for a row query need not be the columns to which you applied predicates prior to calling the `select` method. The columns that you select are the columns of data that you want to access from rows in the row query result.

Names for selected columns

You can always access values for selected columns in a row query result by their position in the list of columns. Alternatively, you can access values by using aliases that you provide when you select the columns. An *alias* is a `String` value of your choice, which serves as a key for accessing specific column values from individual rows in the result. If you do not specify an alias, the default name of a column has the `entity.property` syntax. An alias replaces this default name.

Aggregate functions in row queries

You set up an *aggregate row query* by applying database aggregate functions to selected columns. For an aggregate row query, the result contains values that are aggregated from the table rows used in the query instead of the rows themselves. The database performs the aggregation as part of executing the query.

Selecting columns for row queries

Select the columns for row queries by passing a variable length list of column selection arguments to the `select` method on query objects. Whenever you pass column selection arguments to the `select` method, you set up a row query. Only the `select` method on query objects can set up a row query. The `select` methods on table and restriction objects accept only empty argument lists. An empty argument list passed to the `select` method sets up an entity query instead of a row query. The syntax for selecting columns for a row query is:

```
Query.select({Query Select Column[, Query Select Column...]})
```

See also

- "Applying a database function to a column" on page 350
- "Paths" on page 377
- "Aggregate functions in the query builder APIs" on page 385

Creating column specifications

The column selection arguments that you pass to the `select` method for a row query are objects that implement `IQuerySelectColumn`. You use methods on the `QuerySelectColumns` class to create column selection arguments. The columns that you select must be columns in the database. You cannot specify virtual properties, enhancement methods, or other entity methods as column selections. To create a column that you reference by `entity.property` syntax or by position, starting from `0`, you use the `path` method on `QuerySelectColumns`. To create a named column, you use the `pathWithAlias` method.

You use methods on the `gw.api.path.Paths` class to create column paths, which are `gw.api.path.PersistentPath` objects. To specify the path to an individual column, you use the `make` method on `Paths`. To specify the path to a column in a related entity, you provide additional parameters to the `make` method. For example, the first two of the following Gosu expressions select the `ID` column from the `Contact` table. The second expression sets the alias `ContactID` on the column. The third expression creates a column for the user name of the user that created the `Contact` entity instance:

```
QuerySelectColumns.path(Paths.make(Contact#ID))
QuerySelectColumns.pathWithAlias("ContactID", Paths.make(Contact#ID))
QuerySelectColumns.path(Paths.make(Contact#CreateUser, User#Credential, Credential#UserName))
```

The following Gosu code sets up a row query that returns the columns `Subtype`, `FirstName`, and `LastName` for all `Person` entity instances and prints the values for each row.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var query = Query.make(Person)

var results = query.select({
  QuerySelectColumns.path(Paths.make(Person#Subtype)),
  QuerySelectColumns.pathWithAlias("FName", Paths.make(Person#FirstName)),
  QuerySelectColumns.pathWithAlias("LName", Paths.make(Person#LastName))
})

for (row in results) {
  // Access the first column by entity.property, the second column by alias, the third by position
  print(row.getColumn("Person.Subtype") + ":\t" + row.getColumn("FName") + " " + row.getColumn(2))
}
```

Using foreign key properties and type key properties in row queries

Selecting a foreign key or type key property as a column in a row query does not provide access to the properties of the entity instance or type code. If you select a column that is a foreign key to a record in another database table, the column provides the identifier value of that record. The type of the foreign key column is `gw.pl.persistence.core.Key`. If you select a column that is a type key, the column provides the value of the type code. The type of the type key column is the corresponding typekey class. For example, the type of an address type typekey is `typekey.AddressType`.

The following Gosu code sets up a row query that returns the columns `AddressType` and `CreateUser` for all `Address` entity instances and prints the values for each row.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

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

```
var results = query.select({
  // The Type column has a type of typekey.AddressType
  QuerySelectColumns.pathWithAlias("Type", Paths.make(Address#AddressType)),
  // The User column has a type of gw.pl.persistence.core.Key
  QuerySelectColumns.pathWithAlias("CUser", Paths.make(Address#CreateUser))
})

for (row in results) {
  // Access the type code value and the entity instance ID
  print(row.getColumn("Type") + " " + row.getColumn("CUser"))
}
```

The output from this code looks like the following lines.

```
business null
business 3
...
home 9
...
```

See also

- “Transforming results of row queries to other types” on page 353
- “Accessing data from virtual properties, arrays, or keys” on page 354

Applying a database function to a column

You often create row queries because you want to apply a database function, such as `Sum` or `Max` aggregates, to the values in a column for selected rows. The following SQL statement is a query that uses a maximum aggregate:

```
SELECT Country, MAX(CreateTime)
  FROM pc_Address
  GROUP BY Country;
```

To create a positional column that you reference by number, starting from `0`, you use the `path` method on `QuerySelectColumns`. To create a named column, you use the `pathWithAlias` method. You use methods on the `DBFunction` class to apply a database aggregate function to a column. For example, the following Gosu expressions both select the maximum value in the `CreateTime` column from the `Address` table. The second expression gives the alias `NewestAddr` to the column.

```
QuerySelectColumns.dbFunction(DBFunction.Max(Paths.make(Address#CreateTime)))
QuerySelectColumns.dbFunctionWithAlias("NewestAddr", DBFunction.Max(Paths.make(Address#CreateTime)))
```

The database performs the aggregation as part of executing the query.

The query builder API inserts a `GROUP BY` clause in the generated SQL if necessary. The arguments to the `GROUP BY` clause are the non-aggregate columns that you provide to the `select` method. If you specify no non-aggregate columns, the query builder does not add a `GROUP BY` clause.

The database aggregate functions on the `DBFunction` class are `Sum`, `Max`, `Min`, `Avg`, and `Count`. The `DBFunction` class does not provide `First` or `Last` functions. The equivalent of the `First` function is the `FirstResult` property of the results that the `select` method returns. You can use the `FirstResult` property to simulate the `Last` function by applying the `orderByDescending` method on the results. Using `FirstResult` provides better query performance than stopping a query iteration after the first row.

Using an aggregate function as a column selection

The following Gosu code uses the `Max` database aggregate function. The query builder API appends a `GROUP BY Country` clause to the SQL query that the database receives.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths
uses gw.api.database.DBFunction
```

```
var query = Query.make(Address)
var latestAddress = QuerySelectColumns.dbFunctionWithAlias("LatestAddress",
       DBFunction.Max(Paths.make(Address#CreateTime)))
var rowResults = query.select({
  QuerySelectColumns.pathWithAlias("Country", Paths.make(Address#Country)),
  latestAddress
})
```

The query returns the time of the most recent address creation for each country.

See also

- "Aggregate functions in the query builder APIs" on page 385

Using a distance function as a column selection

The following two Gosu code examples use the `Distance` database function.

The first code example constructs a query that returns all unique addresses within ten miles of San Mateo along with their respective distances from San Mateo in miles. The code example then prints the results of the query.

In particular, the code example sets up a row query, `addressQuery`. For all distinct `Address` entity instances in the database, this row query returns the following columns: `Line1`, `City`, `State`, `PostalCode`, and `Distance`. Constructing the last column, `Distance`, involves calling the `Distance` method.

The `Distance` method returns a simple function. This function returns the distance between the location of an entity instance and a common reference location.

As a first parameter, the `Distance` method takes a row query with a proximity or spatial restriction. Due to a call to the `withinDistance` method, the `addressQuery` row query qualifies as having such a restriction.

As a second parameter, the `Distance` method takes the `String` name of a location property. The `addressQuery` row query rows represent `Address` entity instances. These entity instances all have a `SpatialPoint` location property. Hence, `"Address.SpatialPoint"` is the `String` name of a location property.

The `Distance` method uses these two parameters to calculate the distance between each location that the `SpatialPoint` property values represent and the reference location for the restriction, San Mateo. The resulting distances supply the `Distance` column with values. The first code example is as follows:

```
uses gw.api.database.DBFunction
uses gw.api.path.Paths
uses gw.api.database.spatial.SpatialPoint
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns

// Make a query on the Address entity. Remove duplicate entities from the query.
var addressQuery = Query.make(Address).withDistinct(true)

// Define the location of the center point for the distance calculation.
var SAN_MATEO = new SpatialPoint(-122.300, 37.550)

// Specify the spatial point in the address instance and the maximum distance from the center.
addressQuery.withinDistance(Address.SPATIALPOINT_PROP.get(), SAN_MATEO, 10, UnitOfDistance.TC_MILE)

// Define the columns for the row query.
var result = addressQuery.select({
  QuerySelectColumns.pathWithAlias("Line1", Paths.make(Address#AddressLine1)),
  QuerySelectColumns.pathWithAlias("City", Paths.make(Address#City)),
  QuerySelectColumns.pathWithAlias("State", Paths.make(Address#State)),
  QuerySelectColumns.pathWithAlias("PostalCode", Paths.make(Address#PostalCode)),
  QuerySelectColumns.dbFunctionWithAlias("Distance", DBFunction.Distance(addressQuery, "Address.SpatialPoint"))
})

// Print the results of the query.
for (address in result) {
  print(address.getColumn("Line1") + ", "
    + address.getColumn("City") + ", "
    + address.getColumn("State") + ", "
    + address.getColumn("PostalCode")
    + " is " + address.getColumn("Distance") + " miles away from San Mateo")
}
```

The second code example constructs a row query, `personQuery`. This query returns the name, primary address, and distance from San Mateo for all people in the database having a primary address that is within ten miles of San Mateo. The code example then prints the results of the query.

The call to the `Distance` method takes the row query, `personQuery`, as a first parameter. As a second parameter, the `Distance` method takes the `String` name of the location property that pinpoints the primary address of each identified `Person`. This `String` name is "`Person.PrimaryAddress.SpatialPoint`." The second code example is as follows:

```
uses gw.api.database.DBFunction
uses gw.api.path.Paths
uses gw.api.database.spatial.SpatialPoint
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns

// Make a query on the Person entity.
var personQuery = Query.make(Person)

// Explicitly join the Address entity for the primary address to the Person.
var addressTable = personQuery.join(Person#PrimaryAddress)

// Define the location of the center point for the distance calculation.
var SAN_MATEO = new SpatialPoint(-122.300, 37.550)

// Specify the spatial point in the person's primary address and the maximum distance from the center.
addressTable.withinDistance(Address.SPATIALPOINT_PROP.get(), "Person.PrimaryAddress.SpatialPoint", SAN_MATEO, 10,
UnitOfDistance.TC_MILE)

// Define the columns for the row query.
var result = personQuery.select({
  QuerySelectColumns.path(Paths.make(Person#FirstName)),
  QuerySelectColumns.path(Paths.make(Person#LastName)),
  QuerySelectColumns.pathWithAlias("Line1", Paths.make(Person#PrimaryAddress, Address#AddressLine1)),
  QuerySelectColumns.pathWithAlias("City", Paths.make(Person#PrimaryAddress, Address#City)),
  QuerySelectColumns.pathWithAlias("State", Paths.make(Person#PrimaryAddress, Address#State)),
  QuerySelectColumns.pathWithAlias("PostalCode", Paths.make(Person#PrimaryAddress, Address#PostalCode)),
  QuerySelectColumns.dbFunctionWithAlias("Distance", DBFunction.Distance(personQuery,
"Person.PrimaryAddress.SpatialPoint"))
})

// Print the results of the query.
for (address in result) {
  print(
    address.getColumn("Person.FirstName") + " "
      + address.getColumn("Person.LastName") + ", "
      + address.getColumn("Line1") + ", "
      + address.getColumn("City") + ", "
      + address.getColumn("State") + ", "
      + address.getColumn("PostalCode")
      + " is " + address.getColumn("Distance") + " miles away from San Mateo")
}
```

Creating and using an SQL database function

The `Expr` method on the `DBFunction` class 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 the order specified to form an SQL expression.

For example, the following Gosu code creates a new database function from column references to two columns, with the sum (+) operator. You can use the new function to compare values against the sum of these two columns.

```
// Create an SQL function that subtracts two integer properties
var query = Query.make(Person).withLogSQL(true)
var expr = DBFunction.Expr({
  query.getColumnRef("NumDependents") , " - ", query.getColumnRef("NumDependentsU18")
})
query.compare(Person#NumDependents, Relop.NotEquals, null)
query.compare(Person#NumDependentsU18, Relop.NotEquals, null)
query.compare(expr, GreaterThan, DBFunction.Constant(2))
```

```
var results = query.select()
results.orderBy(QuerySelectColumns.path(Paths.make(Person#FirstName)))
        .thenBy(QuerySelectColumns.path(Paths.make(Person#LastName)))

print("Rows where Person has more than 2 dependents aged 18 or over")
for (row in results) {
  print(row.DisplayName + " Total Dependents " + row.NumDependents +
          ", Dependents 18 or over " + (row.NumDependents - row.NumDependentsU18))
}
```

This code prints results similar to the following:

```
Rows where Person has more than 2 dependents aged 18 or over
Adam Auditor Total Dependents 5, Dependents 18 or over 3
Alex Griffiths Total Dependents 4, Dependents 18 or over 3
Alice Shiu Total Dependents 5, Dependents 18 or over 5
Allison Whiting Total Dependents 6, Dependents 18 or over 5
Annie Turner Total Dependents 5, Dependents 18 or over 3
...
```

Transforming results of row queries to other types

The default type of a row query result is an instance of a class that implements `IQueryResult`. This object provides zero or more row objects of type `QueryRow<Object>`. The `IQueryResult` object is iterable. You convert this result to another type for special purposes, such as using a row query result as the data source for a list view in a page configuration file.

You can specify whatever return type is most convenient for your program. The type does not need to match the native types for the columns in the database.

IMPORTANT Choose the type that you want for the result carefully. Performance characteristics vary depending on the type that you choose.

Transforming row query results to Gosu or Java objects

You can use the `transformQueryRow` method on the result object of a row query to obtain a query result of Gosu or Java objects. For example, you can transform multiple columns to a Gosu data transfer object or a single column to a Java `String`. The return value of `transformQueryRow` is an instance of a class that implements `IQueryResult`. This object provides zero or more objects of the transformed type. For example, the following Gosu code returns an `IQueryResult` object that contains objects of type `ShortContact`.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var query = Query.make(Person).withLogSQL(true)

var results = query.select({
        QuerySelectColumns.pathWithAlias("Subtype", Paths.make(Person#Subtype)),
        QuerySelectColumns.pathWithAlias("PName", Paths.make(Person#FirstName))
  }).transformQueryRow(\row -> {
    // Create a ShortContact object for each row
    return new ShortContact (
            row.getColumn("Subtype") as typekey.Contact, row.getColumn("PName") as String)
})

for (p in results index i) {
  print ((i +1) + ": " + p.Subtype.Description + " " + p.Name)
}

class ShortContact {
  var _subType : typekey.Contact as Subtype
  var _name : String as Name
  construct(st : typekey.Contact, n : String){
    _subType = st
    _name = n
```

```
  }
}
```

Sometimes you only need a single column from each entity. In these cases, you use a simpler syntax for the argument to `transformQueryRow`. For example, to change the preceding code to retrieve just the `PName` column instead of a `ShortContact` object, use the following line that returns an `IQueryResult<Person, String>` object.

```
transformQueryRow(\row -> row.getColumn("PName") as String)
```

Transforming row query results to collections and lists

The result of a row query returned by the select method with parameters is an `IQueryResult`. A row query result is iterable, so you can use it directly in a `for` loop without transformation.

If you need a different iterable type, you can convert the query result objects of row queries to lists, arrays, collections, and sets by using the conversion methods:

- `toCollection`
- `toList`
- `toSet`
- `toTypedArray`

For example, you can use a conversion method on the returned value from the `transformQueryRow` method to produce a `Collection` of data transfer objects.

These conversion methods run the query and materialize the results into memory. If your result set is large, running these methods can exceed the available memory. For large result sets, best practice is to set the page size to prefetch query results.

The `where` method is not applicable on lists, arrays, collections, or sets returned by conversion methods on result objects. Instead, apply all selection criteria to query objects by using predicate methods before calling the `select` method.

See also

- “Blocks” on page 175
- “Using generics with collections and blocks” on page 228
- “Setting the page size for prefetching query results” on page 373

Accessing data from virtual properties, arrays, or keys

A row query provides access to specific properties of primary and secondary entities that are backed by columns in the database.

A row query cannot directly access virtual properties. The values of virtual properties are provided by methods rather than directly from the database.

A row query can access secondary entities from foreign keys or edge foreign keys. A row query cannot directly access arbitrary properties of secondary entities from edge foreign keys or foreign keys. A row query can provide only the identifier value that these properties contain or specified properties of the secondary entity.

A row query cannot access array properties or one-to-one properties. A row query cannot directly access any properties of secondary entities from array properties or one-to-one properties.

The most straightforward way to access these types of values is to use an entity query instead of a row query.

Alternatively, you can use a bundle. This approach performs a query on the database for each row in the result set to retrieve the necessary entities. The following code demonstrates the use of a bundle and keys to access a virtual property and entities in an array.

```
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths
uses gw.pl.persistence.core.Key

// Get the current bundle from the programming context.
var bundle = gw.transaction.Transaction.getCurrent()
```

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

 var results = query.select({
  QuerySelectColumns.pathWithAlias("CUser", Paths.make(Address#CreateUser)),
  QuerySelectColumns.path(Paths.make(Address#PublicID))
})

 for (row in results) {
  if (row.getColumn("CUser") != null) {
    var user = bundle.loadBean(row.getColumn("CUser") as Key)
    print("User login name: " + (user as User).Credential.UserName + " created address: "
            + row.getColumn("Address.PublicID"))
  }
}
```

To run the preceding code in the Gosu Scratchpad, use the method `Transaction.runWithNewBundle` instead of `Transaction.getCurrent`. Enclose the remaining code in a code block, as shown in the following code.

```
gw.transaction.Transaction.runWithNewBundle( \ bundle -> {
  var query = Query.make(Address)
...
}, "su")
```

On your production system, Guidewire strongly recommends that you do not use the Guidewire sys user (System User), or any other default user, to create a new bundle.

See also

- "Building a simple query" on page 314

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 values of columns and foreign keys with row queries. To access values of virtual properties, or properties of entities through arrays or one-to-ones, you must use entity queries, additional queries, or bundles.
- **Result values are not in the application cache** – Values in the result are in local memory only. The application cache does not contain the results.
- **Results cannot be updated** – You cannot update the database with changes that you make to data returned by a row query.
- **Incompatible with list views or detail panels** – The columns in a row in a row query result do not have properties that the PolicyCenter user interface can access. You must transform the row query results to another type before using them as data sources for list views or detail panels in page configuration files.

See also

- "Transforming results of row queries to other types" on page 353
- "Comparison of entity and row queries" on page 357
- "Performance differences between entity and row queries" on page 370

Limitations of MonetaryAmount objects in aggregate queries

The `MonetaryAmount` object encapsulates two main properties: `Amount`, a `BigDecimal` that contains a numeric value, and `Currency`, a `Currency` object. You can use `MonetaryAmount` objects in ordering, column comparisons, and constant comparisons.

If you want to use aggregate functions such as `Avg`, `Count`, `Min`, `Max`, and `Sum`, be aware of the following:

- `MonetaryAmount` objects cannot be used in aggregate functions.
- The individual properties `MonetaryAmount.Amount` and `MonetaryAmount.Currency` are normal properties for the query builder APIs. For example, you can use `Sum` on the `MonetaryAmount.Amount` property, or use `Count` on the `MonetaryAmount.Currency` property.

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. If you expect that the query will return more results than the user interface can display in a single page, you can set the page size to prefetch query results. Setting the page size is also useful if you want to use only the first few rows of a result set.

See also

- "Setting the page size for prefetching query results" on page 373

What result objects contain

The query builder API supports 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 by passing a Gosu array of column selections to the `select` method. For example:

```
query.select({
  QuerySelectColumns.path(Paths.make(Person#Subtype)),
  QuerySelectColumns.pathWithAlias("FName", Paths.make(Person#FirstName)),
  QuerySelectColumns.pathWithAlias("LName", Paths.make(Person#LastName))
})
```

Contents 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 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 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 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 array 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 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 array that you pass to the `select` method. To set up the preceding query as a row query, call the `select` method and pass a Gosu array that specifies the columns you want to select for the result.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var query = Query.make(Company)
var rowResult = query.select({ // The select method with an array argument sets up a row query.
  QuerySelectColumns.pathWithAlias("Name", Paths.make(Company#Name)),
  QuerySelectColumns.pathWithAlias("City", Paths.make(Company#PrimaryAddress, Address#City)),
  QuerySelectColumns.path(Paths.make(Company#PrimaryAddress, Address#Country))
})
```

The members of results from row queries are structures that correspond to the column selection array that 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 code prints the company name and the city of the primary address for each `Company` in the result. The result member provides the `Name` column and the `City` column.

```
for (company in rowResult) {  // Members of a result from a row query are QueryRow objects.
  print (company.getColumn("Name") + ", " +
         company.getColumn("City") + ", " +
         company.getColumn(2))
}
```

You can only access columns from the members of a row result if you specified them in the column selection array.

See also

- “Paths” on page 377

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.

Feature	Entity queries	Row queries
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 directly be data sources for list views or detail panels in page configuration.

See also

- "Performance differences between entity and row queries" on page 370

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 the result. For example, you have a query with complicated joins, predicates, and subselects. Several routines need 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. Building the query once in your main routine improves the process of debugging and maintaining the query logic.

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. PolicyCenter 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 PolicyCenter 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.

See also

- "Using standard query filters in toolbar filters" on page 361

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 Gosu code creates a standard query filter that can apply to query results that include `Address` instances. The standard query filter predicate uses a `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 predefined 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 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 need 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.

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 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 2017 or later.

```
uses gw.api.database.Query
uses gw.api.filters.StandardQueryFilter
uses gw.api.util.DateUtil

// 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(Address#City, Equals, "Chicago")})

// Create a standard query filter for addresses added during 2017 or later.
var queryFilter2017Address = new StandardQueryFilter("2017 Addresses",
        \ query -> {query.compare(Address#CreateTime, GreaterThanOrEquals,
                                  DateUtil.createDateInstance(1, 1, 2017))})

// Add the Chicago and 2017 address filters to the result.
result.addFilter(queryFilterChicago)
result.addFilter(queryFilter2017Address)

// Iterate the addresses in Chicago added during 2016 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 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
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

// 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(Address#City, Equals, "Bloomington")
                                orCriteria.compare(Address#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(QuerySelectColumns.path(Paths.make(Address#City)))) {
  print(address.City + ", " + address.State + "  " + address.PostalCode)
}
```

See also

- “Combining predicates with AND and OR logic ” on page 330

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 if joins, subselects, and compound predicate expressions are involved. Placing this part of your query builder code in one location improves the process of debugging and maintaining your code.

The following Gosu code builds a query for addresses in the state of Illinois, and then 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
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

// Select addresses from the state of Illinois.
var query = Query.make(Address)
query.compare(Address#State, Equals, typekey.State.TC_IL)

// Get a result and apply ordering
var result = query.select()
result.orderBy(QuerySelectColumns.path(Paths.make(Address#City)))
       .thenBy(QuerySelectColumns.path(Paths.make(Address#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",
          \ filterQuery -> {filterQuery.compare(Address#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",
          \ filterQuery -> {filterQuery.compare(Address#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",
          \ filterQuery -> {filterQuery.compare(Address#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 of 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` properties each specify a standard query filter by using an inline constructor. The filter applies to work queue tasks that the list view in the `WorkQueueExecutorsPanelSet` PCF file displays. `WorkQueueExecutorsPanelSet` is a separate PCF file included in `WorkQueueInfo.pcf`. The `TaskFilter` toolbar filter has two `ToolbarFilterOption` filter values:

```
new gw.api.filters.StandardQueryFilter("All", \ q -> {})
new gw.api.filters.StandardQueryFilter("With errors",
        \ q -> q.compare("Exceptions", gw.api.database.Relop.GreaterThan, 0))
```

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 uses the first parameters of the filters to provide two options, “All” and “With errors”, 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 `filters` property combines the two `filter` properties in the `WorkQueueExecutorsPanelSet` PCF file to specify an array of two standard query filters by using inline constructors. This `ToolbarFilterOptionGroup` example replaces the two `ToolbarFilterOption` filter values on the `TaskFilter` toolbar filter:

```
new gw.api.filters.StandardQueryFilter[] {
  new gw.api.filters.StandardQueryFilter("All", \ q -> {}),
  new gw.api.filters.StandardQueryFilter("With errors",
          \ q -> q.compare("Exceptions", gw.api.database.Relop.GreaterThan, 0))}
```

Note: The previous code block for the single-line `filters` 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, “All” and “With errors”, 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.

```
policy.PolicyNumber
```

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 need 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 from the database in no specific order. Results in this apparently random order might not be useful. SQL and the query builder APIs support specifying the order of items in the results.

With SQL, the `ORDER BY` clause specifies 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;
```

The following Gosu code uses the `orderBy` ordering method to sort addresses in the same way as the preceding SQL example.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var queryAddresses = Query.make(Address) // Query for addresses in Chicago.
queryAddresses.compare(Address#City, Equals, "Chicago")
var selectAddresses = queryAddresses.select()

// Sort the result by postal code.
selectAddresses.orderBy(QuerySelectColumns.path(Paths.make(Address#PostalCode)))

var resultAddresses = selectAddresses.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. Gosu does not submit the query to the database until you begin to iterate a result object. 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 API

The Query builder API supports the following ordering methods on result objects.

Method	Description
`orderBy`	Clears all previous ordering, and then orders results by the specified column in ascending order.
`orderByDescending`	Clears all previous ordering, and then orders results by the specified column in descending order.
`thenBy`	Orders by the specified column in ascending order, without clearing previous ordering.
`thenByDescending`	Orders by the specified column in descending order, without clearing previous ordering.

The ordering methods all take an object that implements the `IQuerySelectColumn` interface in the `gw.api.database` package as their one argument. To create this object, use the following syntax.

```
QuerySelectColumns.path(Paths.make(PrimaryEntity#SimpleProperty))
```

The `Paths.make` method yields an object access path from the primary entity to a simple, non-foreign-key, database-backed property. Gosu checks property names against column names in the *Data Dictionary*. To specify a path to a property on a dependent table, specify a foreign-key property to the dependent table on the previous table parameter.

For example, the following Gosu code specifies a simple property, the `PostalCode` on an `Address` instance.

```
QuerySelectColumns.path(Paths.make(Address#PostalCode))
```

See also

- "Paths" on page 377
- "Locale sensitivity for ordering query results" on page 364
- *Globalization Guide*

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. The access path can traverse only database-backed foreign keys, and 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 parameters that you pass to the ordering methods.

The following Gosu code orders notes, based on the date that the activity related to a note was last viewed.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var queryNotes = Query.make(Note)  // Query for notes.
var resultNotes = queryNotes.select()

// Sort the notes by related date on activity.
resultNotes.orderBy(QuerySelectColumns.path(Paths.make(Note#Activity, Activity#LastViewedDate)))
```

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;
```

The following Gosu code constructs and executes a query that is functionally equivalent to the preceding SQL example.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var queryAddresses = Query.make(Address) // Query for addresses.
var resultAddresses = queryAddresses.select()

// Sort results by city.
resultAddresses.orderBy(QuerySelectColumns.path(Paths.make(Address#City)))
// Within a city, sort results by postal code.
resultAddresses.thenBy(QuerySelectColumns.path(Paths.make(Address#PostalCode)))
// Within a postal code, sort results by street.
resultAddresses.thenBy(QuerySelectColumns.path(Paths.make(Address#AddressLine1)))
```

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. In contrast, collection enhancement methods for ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. To sort `String` values in a locale-sensitive way, you supply an optional `Comparator` argument to those methods.

Note: In configurations that use primary or secondary linguistic sort strength, the ordering of results is case-insensitive. If either of your PolicyCenter or the relational database that your PolicyCenter uses have these configurations, the query result includes ignores the case of `String` values. For example, in these configurations, “Building Renovators” and “building renovators” are adjacent in the results.

See also

- “Sorting lists or other comparable collections” on page 190
- “List of enhancement methods on collections” on page 195
- *Globalization Guide*

Useful properties and methods on result objects

Query result objects have these useful properties:

- `Empty` – Informs you whether a query failed to fetch any matching items.
- `Count` – Returns the number of items in the result.
- `AtMostOneRow` – Specifies that you want the query to return a result only if a single item matches the query predicates.
- `FirstResult` – Provides the first item in a result without iterating the result object.
- `getCountLimitedBy` – Informs you whether the result contains more items than you can use.

Determining whether a query returned no results

Queries can return no results under certain conditions, such as queries that include predicates that users provide through the user interface. For example, users can search for people by name by using the user interface, such as finding people 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, as the following Gosu code shows.

```
uses gw.api.database.Query

var query = Query.make(Person)

// Apply some subselect, join, and predicate methods here.

result = query.select()

if (result.Empty) {
  ...
}
```

Alternatively, you can test the `Count` property against zero or iterate a result object inside a `while` loop with a counter and then test the counter for zero. Relational query performance often improves if 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 rows 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. Do not rely on the count number remaining constant. That number might be useful in some contexts such as simple user interface messages such as “Displaying items 1-10 out of 236 results”. However, that count might be different from the number of items returned from a query even if you iterate across the result set immediately after retrieving the count. Other actions may add, change, or remove rows from the database between the time you access the `Count` property and when you iterate across the results.

Unsafe usage:

```
// Bad example. Do NOT follow this example. Do NOT rely on the result count staying constant!

uses gw.api.database.Query

// create a query
var query = Query.make(User)

// THE FOLLOWING LINE IS UNSAFE
var myArray = new User[ query.select().Count ]

for (user in query.select() index y) {
  // This line throws an out-of-bounds exception if more results appear after the count calculation
   myArray[y] = user
}
```

Code like the previous example risks throwing array-out-of-bounds errors at run time. Adding a test to avoid the exception risks losing records from the result.

Instead, iterate across the set and count upward, appending query result entities to an `ArrayList`.

Safe usage:

```
uses gw.api.database.Query
uses java.util.ArrayList

var query = Query.make(User)

// Create a new list, and use generics to parameterize it
var myList = new ArrayList<User>()

for (user in query.select()) {
  // Add a row to the list
  myList.add(user)
}
```

Calling `query.select()` does not snapshot the current value of the result set forever. When you access the `query.select().Count` property, PolicyCenter runs the query but the query results 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 of how many instances the database can provide. Use the `FirstResult` property on a result object to obtain its first item, as the following Gosu code shows.

```
uses gw.api.database.Query

var query = Query.make(Person)

query.compareIgnoreCase(Person#FirstName, Equals, "ray")
query.compareIgnoreCase(Person#LastName, Equals, "newton")

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 only the first item in a result.

Note: To find the last item in a result, reverse the order of the rows by calling the `orderByDescending` method.

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 more than you require, which 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 a hundred items. Pass the number 101 to the `getCountLimitedBy` method, and check the value that the method returns. If the method returns a number less than 101, the result will be within your upper bound, so you can safely iterate the result. If the method returns 101, 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 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(Address#City, Equals, "Chicago")
var result = query.select()

// Specify the threshold for too many items in the result
var threshold = 101  // -- 101 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 the query result objects of entity queries to lists, arrays, collections, and sets by using the conversion methods:

- `toCollection`
- `toSet`
- `toList`
- `toTypedArray`

IMPORTANT Do not use the `where` method on lists, arrays, collections, or sets returned by conversion methods on result objects. Instead, apply all selection criteria to query objects by using predicate methods before calling the `select` method.

Converting a query result to these types executes 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 conversion for queries of unknown size. Only do this conversion 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 conversion 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:

```
uses gw.api.database.Query

// Query for addresses in the city of Chicago
var query = Query.make(Address)
query.compare(Address#City, Equals, "Chicago")
var result = query.select()

// Specify the threshold for too many items in the result
var threshold = 101  // -- 101 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!")
}
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: typeof " + typeof resultCollection + "/ size " + resultCollection.Count)
  print("Set: typeof " + typeof resultSet + "/ size " + resultSet.Count)
  print("List: typeof " + typeof resultList + "/ size " + resultList.Count)
  print("Array: 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: typeof java.util.ArrayList/ size 34
Set: typeof java.util.HashSet/ size 34
List: typeof java.util.ArrayList/ size 34
Array: typeof entity.User[]/ size 34
```

See also

- “Transforming results of row queries to other types” on page 353

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. In many uses of query results, you need to write changed records

to the database. For example, you use custom batch processing to flag contacts that your users need to call the next day.

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 bundler, 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(Address#State, Equals, typekey.State.TC_IL)
var result = query.select().orderBy(QuerySelectColumns.path(Paths.make(Address#City)))

for (address in result) {
  switch (address.City) {
    case "Schaumburg":
    case "Melrose Park":
    case "Norridge":
      break
    default: {
      // Add the address to the current bundle to make it writable and
      // discard the read-only reference obtained from the query result.
      address = bundle.add(address)
      // Change properties on the writable 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.

The Gosu Scratchpad does not have a current bundle, so you must create a new bundle. To run the preceding code in the Gosu Scratchpad, use the method `Transaction.runWithNewBundle` instead of `Transaction.getCurrent`. Enclose the remaining code in a code block:

```
gw.transaction.Transaction.runWithNewBundle( \ bundle -> {
  var query = gw.api.database.Query.make(Address)
...
}, "su")
```

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(Address#State, Equals, typekey.State.TC_IL)

// Configure the result to be ordered by City.
var result = query.select()
result.orderBy(QuerySelectColumns.path(Paths.make(Address#City)))

// Iterate and print the result set.
for (address in result) {
  print(address.City + ", " + address.Description)
}
```

See also

- "Bundles and database transactions" on page 387
- *Integration Guide*

Testing and optimizing queries

The query builder APIs provide multiple options for testing and optimizing queries.

Performance differences between entity and row queries

The query builder API supports 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 by passing a Gosu array of column selections to the `select` method. For example:

  ```
  query.select({
    QuerySelectColumns.path(Paths.make(Person#Subtype)),
    QuerySelectColumns.pathWithAlias("FName", Paths.make(Person#FirstName)),
    QuerySelectColumns.pathWithAlias("LName", Paths.make(Person#LastName))
  })
  ```

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 used in batch processing types.

Use entity queries for code readability and maintenance

Generally, use an entity query unless performance is not adequate. The query builder code for entity queries is more readable and maintainable than for row queries, especially if you access the query builder APIs from Java rather than Gosu.

Use row queries to improve performance

Generally, use a row query instead of an entity query if the performance of the entity query is inadequate. Entity queries sometimes fetch unused data columns or execute additional queries in code that processes the results. Row queries can improve performance in these cases. Navigating object path expressions that span arrays and foreign keys to access the data that you need can cause additional, implicit database queries to fetch the data. Row queries fetch only the data that you need and typically avoid these additional queries.

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.

See also

- "Limitations of row queries" on page 355

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 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 /* pc:T:Gosu class redefiner; */ FROM pc_contact gRoot WHERE gRoot.Retired = 0
```

The first line shows square brackets (`[]`) containing the list of variables to bind to the query. In this case, there are no variables. The remaining lines show the SQL statement. Note that the table for the entity type, `pc_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 an approximation of the SQL statement that PolicyCenter submits to the application database. The actual SQL statement might differ due to database optimizations that PolicyCenter applies internally.

Using withLogSQL to record SQL SELECT statements

You might want to see the underlying SQL `SELECT` statement that the query builder API submits to the application database. Use the `withLogSQL` method on a query to see the statement. When you turn on logging behavior, the query builder API writes the SQL `SELECT` statement to the system logs, in logging category `Server.Database`. The API also writes the SQL statement to standard output (`stdout`).

For example, the following Gosu code turns logging on by calling the `withLogSQL` method of the query object. The SQL statement is written to the system logs at the time code starts to iterate the results.

```
uses gw.api.database.Query

var query = Query.make(Person).withLogSQL(true)
query.startsWith(Person#LastName, "A", false)
query.withLogSQL(true)          // -- turn on logging behavior here --
var result = query.select().orderBy(QuerySelectColumns.path(Paths.make(Person#LastName)))
          .thenBy(QuerySelectColumns.path(Paths.make(Person#FirstName)))
```

```
var i = result.iterator()       // -- write the SQL statement to the system logs here --
while (i.hasNext()) {
  var person = i.next()
  print (person.LastName + ", " + person.FirstName + ": " + person.EmailAddress1)
}
```

The SQL Select statement for the preceding example looks like the following on standard output.

```
Executing sql = SELECT /* KeyTable:pc_contact; */
        gRoot.ID col0, gRoot.Subtype col1, gRoot.LastName col2, gRoot.FirstName col3
        FROM pc_contact gRoot
        WHERE gRoot.Subtype IN (?,?,?,?) AND gRoot.LastName LIKE ? AND gRoot.Retired = 0
        ORDER BY col2 ASC, col3 ASC
        [2 (typekey), 4 (typekey), 7 (typekey), 9 (typekey), A% (lastname)]
```

The statement on your system depends on your relational database and might differ from the preceding example.

Note: Writing to the system logs and to standard output does not occur when Gosu code calls the `withLogSQL` method. That logging occurs some time later, when PolicyCenter submits the query to the relational database.

Enabling context comments in queries on SQL Server

On SQL Server, tuning queries can be difficult because you cannot determine what part of the application generates specific queries. To help tune certain queries, you can enable two configuration parameters in `config.xml`.

- `IdentifyQueryBuilderViaComments` – Instrumentation comments from higher level database objects constructed by using the query builder APIs
- `IdentifyORMLayerViaComments` – Instrumentation comments from lower level objects, such as beans, typelists, and other database building blocks

If you set either parameter to true, PolicyCenter adds SQL comments with contextual information to certain SQL `SELECT` statements that it sends to the relational database. The default for `IdentifyQueryBuilderViaComments` is `true`. The default for `IdentifyORMLayerViaComments` is `false`.

The SQL comments are in the format:

```
/* applicationName:ProfilerEntryPoint */
```

The *`applicationName`* component of the comment is `PolicyCenter`.

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, call the `withFindRetired` method with the value `true`, as the following Gosu code shows.

```
uses gw.api.database.Query

var query = Query.make(Activity)
query.compare(Activity#ActivityPattern, Equals, null)
query.withFindRetired(true)
```

IMPORTANT Use the `withFindRetired` method on queries only under exceptional circumstances that require exposing retired entities.

Including temporary branches in EffDated query results

In user-interface interactions, when a new job is created, a temporary branch, or policy graph, is created in the database. PolicyCenter adjusts the branch to represent the initial state of the new job, removes the temporary flag, and commits the changes. These temporary branches exist for a very brief time and have little, if any, value to processes other than that which is creating the new job.

You may have custom code that create new jobs. Examples include integration- or spreadsheet-driven import and migration of policy data. These processes create temporary branches in the database, just like user-interface interactions.

Your process may need to access and change the `EffDated` entities in the newly created temporary branch. For example, your process adjusts the data by normalizing upper and lower case in certain fields. To access the `EffDated` entities on the temporary branch, call the `withFindTemporaryBranches` method on the query with value `true`.

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 a batch into a cache in the application server for quick access. Common actions like iterating across the query have higher performance by using this cache.

In SQL, this caching 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*.

Using only the first page of the results is equivalent to the SQL `TOP` or `LIMIT` functionality.

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:

```
uses gw.api.database.Query

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, you must use a transaction.
- 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.

See also

- “Updating entity instances in query results” on page 368

Using settings for case-sensitivity of text comparisons

Upper and lower case lettering affect comparison of written English values during query selection and result ordering. Often, you want to ignore the difference between “A” and “a” when queries select text values.

Languages other than English have other character modifications that affect comparison for selection and ordering. For example, many written European languages have characters with accents and other diacritical marks on a base letter. Sometimes you want to ignore the differences between base letters with and without diacritics, such as the differences between “A”, “À”, and “Á”. Asian languages have other concerns, such as single-byte and double-byte characters or between katakana and hirigani characters in Japanese.

The result of comparing the value of a character field with another character value differs depending on the language, search collation strength, and database that your PolicyCenter uses. For example, case-insensitive comparisons produce the same results as case-sensitive comparisons if PolicyCenter has a linguistic search strength of primary.

Some query API methods support specifying whether to ignore differences between letter case. When you specify that you want to ignore case, PolicyCenter uses your localization settings to control the results. For each locale configured in your PolicyCenter instance, you specify which dimensions of difference you want to ignore. For example, you can configure a locale to ignore case only, to ignore case and accents, or to ignore other dimensions of difference that are relevant to the locale.

Query performance can suffer when you ignore case in query predicate methods. To improve query performance if you typically need to ignore case in text comparisons, set the `SupportsLinguisticSearch` attribute on `column` elements in entity definitions. For a column that has the `SupportsLinguisticSearch` set, PolicyCenter creates a corresponding denormalization column in addition to the standard column that stores the field values. When PolicyCenter stores a value in the standard column, PolicyCenter also stores a value in the denormalization column. PolicyCenter saves a denormalized value by converting the regular value to one that ignores character differences, based on the dimensions of difference that you specified in your localization settings. When a query applies a predicate that specifies ignoring the case of a field, PolicyCenter uses the same algorithm that converted values to store in the denormalization column. The relational database compares the converted bound value to the values in the denormalization column, not the values in the standard column.

For example, you specify the following query:

```
uses gw.api.database.Query

// Query the Person instances for a specific last name.
var query = Query.make(Person).withLogSQL(true)
query.compare(Person#LastName, Equals, "smith")
```

Print the rows that the query returns by adding the following lines:

```
// Fetch the data and print it.
var result = query.select()
for (person in result) {
  print (person.DisplayName)
}
```

Gosu creates a query that looks like the following one.

```
SELECT /* KeyTable:pc_contact; */
  gRoot.ID col0,
  gRoot.Subtype col1
FROM pc_contact gRoot
WHERE gRoot.LastName = ?
AND gRoot.Retired = 0 [smith (lastname)]
```

Depending on your PolicyCenter and database settings, you see either no output or output that looks like:

```
Steve Smith
Kerry Smith
Alice Smith
John Smith
...
```

To do a case-insensitive text comparison, use the `compareIgnoreCase` method. This method uses the same parameters as the `compare` method. To see the effect on the SQL query of using the `compareIgnoreCase` method, change the comparison line in the code to:

```
query.compareIgnoreCase(Person#LastName, Equals, "smith")
```

Gosu creates a query that looks like the following one.

```
SELECT /* KeyTable:pc_contact; */
  gRoot.ID col0,
  gRoot.Subtype col1
FROM pc_contact gRoot
WHERE gRoot.LastNameDenorm = ?
AND gRoot.Retired = 0 [smith (lastname {linguistic=true})]
```

The standard column has values like “Smith” and “Menzies”. The denormalization column has corresponding values like “smith” and “menzies”. The preceding query returns `Person` entity instances with the last name “Smith” in the standard column, because it compared the bound value “smith” to the value in the denormalization column.

You see output that looks like:

```
Steve Smith
Kerry Smith
Alice Smith
John Smith
...
```

See also

- *Globalization Guide*
- *Configuration Guide*

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 object type that a chain of methods returns is the return type of the final method in the chain. If you need to use a particular object type as an argument to a function, you must ensure that the object that the chain returns is the correct type.

The following Gosu code places each query builder API call in a separate statement. The object type of `queryPerson` is `Query<Person>`. The object type of `tableAddress` is `Table<Person>`. The return value of the `contains` method, a `Restriction<Person>` object, is not used. The object type of `result` is `IQueryBeanResult<Person>`.

```
uses gw.api.database.Query

// Query builder API method calls as separate statements
var queryPerson = Query.make(Person)
var tableAddress = queryPerson.join(Person#PrimaryAddress)
tableAddress.contains(Address#AddressLine1, "Lake Ave", true)

var result = queryPerson.select()

// Fetch the data with a for loop
for (person in result) {
  print (person + ", " + person.PrimaryAddress.AddressLine1)
}
```

The following Gosu code is functionally equivalent to the sample code above, but it uses method chaining to condense the separate statements into one. Because the code uses only the value of result, an `IQueryBeanResult<Person>` object, none of the intermediate return objects are necessary.

```
uses gw.api.database.Query

// Query builder API method calls chained as a single statement
var result = Query.make(Person).join(Person#PrimaryAddress).contains(Address#AddressLine1,
        "Lake Ave", true).select()
```

```
// Fetch the data with a for loop
for (person in result) {
  print (person + ", " + person.PrimaryAddress.AddressLine1)
}
```

When you chain methods, the objects that support the calls in the chain often do not appear explicitly in your code. In the example above, the `Query.make` method returns a query object, on which the chained statement calls the `join` method. In turn, the `join` method returns a table object, on which the next method calls the `contain` method. The `contain` method returns a restriction object, which has a `select` method that returns a result object for the built-up query. After the single-line statement completes, the query object is discarded and is no longer accessible to subsequent Gosu code.

Note: 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 chained query builder statement that returns a result object.

Working with nested subqueries

A *nested subquery* simplifies a complex query by mapping a set of columns to a pseudo-table. You define the pseudo-table as a query that maps to an alias name in the `FROM` clause of an SQL query. The parent table in the SQL query joins to the pseudo-table in the same way as it joins to any other secondary table. A typical use of a nested subquery is when you need to access an aggregate value from a secondary table. This advanced feature is not normally needed in production code. Use this feature only if necessary due to the performance implications. For more information about nested subqueries, check the documentation for your database. The Oracle term for a nested subquery is an *inline view*. The SQL Server term for a nested subquery is a *derived table*. If you have questions about the usage of nested subqueries, contact Guidewire Customer Support.

For example, the following SQL query uses a nested subquery to compare the minimum value of a secondary table column with the value of a primary table column:

```
SELECT *
  FROM pc_Address
  INNER JOIN ( SELECT Country, MIN(City) MinCity
            FROM pc_Address
            GROUP BY Country ) MinCityAddress
    ON pc_Address.City = MinCityAddress.City;
```

To create a nested subquery, call the `inlineView` method on the query. This method returns a query using the new nested subquery and includes all referenced columns from that query in the `select` statement. The `inlineView` method takes the following arguments:

- `joinColumnOnThisTable` – The name of the join column (`String`).
- `inlineViewQuery` – The query (`Query`). This argument cannot be the result of any `Query` method that returns a `Table`.
- `joinColumnOnViewTable` – The name of the join column on the nested query table (`String`).

The method returns a new query that contains the nested subquery. The return type of the method is a `Table` object. Predicates on the secondary table can use columns on the secondary table or the primary table.

For example, suppose you create two queries, an inner query and an outer query:

```
var innerQuery = Query.make(Address)
var outerQuery = Query.make(Address).withLogSQL(true)
```

Next, create a nested subquery from the inner query to add the `Address#Country` column to the columns that the outer query returns:

```
var nestedQuery = outerQuery.inlineView("Country", innerQuery, "Country")
```

Next, use the `City` column in a new predicate on the outer query:

```
outerQuery.compare(Address#City, Equals,
          nestedQuery.getColumnRef(DBFunction.Min(Paths.make(Address#City))))
```

Test the code:

```
for (row in outerQuery.select()){
  print(row.DisplayName)
}
```

This code prints the display name of any `Address` entity that has a city that matches the city that is alphabetically first for a particular country. This produces the following SQL statement:

```
SELECT /* KeyTable:pc_address; */ gRoot.ID col0, gRoot.Subtype col1
  FROM pc_address gRoot
  INNER JOIN (
    SELECT address_0.Country col0, MIN(address_0.City) col1
      FROM pc_address address_0 WHERE address_0.Retired = 0
      GROUP BY address_0.Country) address_0
    ON gRoot.Country = address_0.col0
  WHERE gRoot.City = address_0.col1 AND gRoot.Retired = 0 []
```

Paths

A path is essentially a type-safe list of property references encapsulated in an object of type `gw.api.path.Path`. Each element of the path must have an owning type that is the same type or subtype of the feature type of the immediately previous element in the path. In other words, starting at element 2, the left side of the literal must match the right side of the literal in the previous element in the path.

From a programming perspective, `Path` is an interface. For use with the query builder APIs, the only relevant implementing class is `PersistentPath`, which contains only persistent property references.

Making a path

To create a path, use the `gw.api.path.Paths` class, which has a static method called `make`. Pass each property reference in order as separate ordered method arguments. The `make` method has method signatures that support paths of length 1, 2, 3, 4, and 5. The `make` method only works with property references that represent persistent properties. A *persistent property* is a property that directly maps to a database field.

The following example creates a path with two property references

```
var path = Paths.make(User#Contact, UserContact#PrimaryAddress)
```

Appending a path

You can append a property with the `append` method:

```
var pathPostalCode = path.append(Address#PostalCode)
```

Converting a path to a list

You can convert a path to an immutable list that implements `java.util.List`:

```
var pathAsList = pathPostalCode.asList()
```

The elements of the list are `Contact`, `PrimaryAddress`, and `PostalCode`.

Getting the leaf value from a path and an object

Given an object and a path, you can get the leaf value (the final and rightmost property in the path) using the `getLeafValue` method:

```
var postalCode = pathPostalCode.getLeafValue(user)
```

You must ensure that no element in the path to the leaf is `null`.

An entity query on the `User` table can use this method to access the leaf values:

```
var query = Query.make(User)
var rowResult = query.select()
for (user in rowResult){
  print(user.DisplayName + " " +
          (user.Contact.PrimaryAddress == null ? "No address" : pathPostalCode.getLeafValue(user)))
}
```

Types and methods in the query builder API

The query builder API in the `gw.api.database` package includes Java classes and interfaces and Gosu enhancements. You access the API in your Gosu code in the same way as any other Gosu API. The query builder API provides types and their methods for building queries and for accessing the rows and columns in the query results. The API also includes types and methods for constructing method parameters. The context-sensitive editor in Studio provides quick access to all these types and methods.

The Javadoc for the Java classes and interfaces in the `gw.api.database` package is available in `PolicyCenter/javadoc`. You can generate and view the Gosudoc that describes Gosu enhancements to the API by running the `gwb gosudoc` command. This command produces documentation at `PolicyCenter/build/gosudoc/index.html`.

See also

- "Gosu generated documentation (Gosudoc)" on page 46

Types and methods for building queries

The following sections describe the major classes, interfaces, and methods that you use to build queries.

Types for building queries

The following Gosu types in the `gw.api.database` package provide methods for building a query. `Query`, `Restriction`, and `Table` all implement the `ISelectQueryBuilder` interface. `IQueryBeanResult` implements `IQueryResult.`

Type	Description
`Query`	A class that represents a query that fetches entities or rows from the application database.
`Restriction`	An interface that represents a Boolean condition or conditions that restricts the set of items that a query fetches from the application database.
`Table`	An interface that represents an entity type that you add to the query with a join or a subselect.
`IQueryBeanResult`	An interface that represents the results of a query that fetches entities from the application database. Adding sorting or filters to this item affects the query.
`IQueryResult`	An interface that represents the results of a query that fetches rows from the application database. Adding sorting or filters to this item affects the query.

Methods on Query type for building queries

The following methods provide fundamental functionality for building queries. Other methods provide filtering to the query. To see information about methods that provide additional functionality, see the Javadoc. You can chain many of these methods to create concise code. For example, you can specify a query on an entity type, join to another entity type, and create the result set object by using a line like the following one:

```
var result = Query.make(Company).join(Company#PrimaryAddress).select()
```

Method	Description	Returned object type
`join`	Returns a table object with information from several entity types joined together in advanced ways.	`Table`
`make`	Static method on the `Query` class that creates a new query object.	`Query`
`select`	Defines the items to fetch from the application database, according to restrictions added to the query. Returns a result object that provides one of the following types of item: • Items fetched from a single, root entity type • Data rows containing specified fields from one or more entity types	`IQueryBeanResult` or `IQueryResult`

`subselect`	Joins a dependent source. For example:
	`var outerQuery = Query.make(User) // Returns User Query`
	`var innerQuery = Query.make(Note) // Returns Note Query`
	`// Filter the inner query`
	`noteQuery.compareIn(Note#Topic, {NoteTopicType.TC_GENERAL, NoteTopicType.TC_LEGAL})`
	`// Filter the outer query by using a subselect`
	`userQuery.subselect(User#ID, InOperation.CompareIn, noteQuery, Note#Author )`
		`Table`

`union`	Combines all results from two queries into a single result.	`GroupingQuery`
`withDistinct`	Whether to remove duplicate entity instances from the result.	`Query`

Methods on Result types for building queries

The following methods provide ordering functionality for building queries.

Other methods provide filtering to the result set. To see information about methods that provide additional functionality, see the Javadoc.

Method	Description
`orderBy`	Clears all previous ordering, and then orders results by the specified column in ascending order.
`orderByDescending`	Clears all previous ordering, and then orders results by the specified column in descending order.
`thenBy`	Orders by the specified column in ascending order, without clearing previous ordering.
`thenByDescending`	Orders by the specified column in descending order, without clearing previous ordering.

These ordering methods all take an object that implements the `IQuerySelectColumn` interface as their one argument.

See also

• "Ordering results" on page 363
• "Locale sensitivity for ordering query results" on page 364

Column selection types and methods in the query builder APIs

The query builder APIs provide multiple ways to access columns in defining a query and in accessing the column values in the result set. The methods that you use depend on the task that you are performing.

Types for column specification in building a query

The query that you specify by using the `Query.make` method provides access to the properties on the entity type of the query. For example, the following line specifies a query on the `Company` entity type:

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

The `PropertyReference` class provides type-safe access to the properties on the query entity type. You specify a property reference by using the following syntax:

```
EntityType#Property
```

You can use a property reference to filter query results, join to another entity type, order rows in the result set, or specify a column for a row query. For example, the following line joins the company query to the `Address` entity type and returns a `Table` object:

```
var tableAddress = queryCompany.join(Company#PrimaryAddress)
```

A `Table` object provides the same type-safe access to properties on its entity type as a `Query` object does. For example, the following lines provide two predicates, one on the query object and one on the table object. These predicates filter the SQL query that is sent to the database to return only companies with the name “Stewart Media” that have a primary address in Chicago.

```
queryCompany.compare(Company#Name, Equals, "Stewart Media")
tableAddress.compare(Address#City, Equals, "Chicago")
```

If you have a query that joins entity types, the `Query` and the `Table` object can perform type-safe access only for properties on their own entity type. If you need to compare property values with each other or filter properties on multiple joined tables to create an `OR` filter, you must use a `ColumnRef` object. For example, the following lines create two predicates on the `Table` object. These predicates filter the SQL query that is sent to the database to return companies with either the name “Stewart Media” or a primary address in Chicago.

```
tableAddress.or( \ or1 -> {
  or1.compare(Address#City, Equals, "Chicago")
  or1.compare(queryCompany.getColumnRef("Name"), Equals, "Stewart Media")
})
```

Some other predicate method signatures take an object that implements the `IQueryablePropertyInfo` interface as a parameter.

Type	Description

`PropertyReference`	A class that provides a type-safe reference to a property. Use this class to access properties on the primary entity of a `Query`, `Table`, or `Restriction`. You can use an instance of this class to create a join or predicate. You can also use an instance of this class to define a query select column to order the query results of for a column in a row query. The following lines use property references to join tables and filter the primary entity of a `Query` and a `Table` object.
	`var queryCompany = Query.make(Company)`
	`var tableAddress = queryCompany.join(Company#PrimaryAddress)`
	`queryCompany.compare(Company#Name, Equals, "Stewart Media")`
	`tableAddress.compare(Address#City, Equals, "Chicago")`

`ColumnRef`	A class that specifies a column reference to a property on a dependent entity, or a comparison property on the primary entity of a `Query`, `Table`, or `Restriction`. A `ColumnRef` instance does not provide type safety. The following lines use a column reference for a property on a dependent table:
	`tableAddress.or( \ or1 -> {`
	`  or1.compare(Address#City, Equals, "Chicago")`
	`  or1.compare(queryCompany.getColumnRef("Name"), Equals, "Stewart Media")`
	`})`

`String`	Some query builder API method signatures accept a `String` argument. If a signature that uses a `PropertyReference` parameter is available, use that type-safe signature instead.

Type	Description

`IQueryablePropertyInfo`	An interface that provides information about a property. Some query builder API method signatures accept an `IQueryablePropertyInfo` argument. To create this object, use code like the following:
	`var lastNamesArrayList = {"Smith", "Applegate"}`
	`var query = Query.make(Person)`
	`query.compareIn(Person.LASTNAME_PROP.get(), lastNamesArrayList)`

Methods for column specification in building a query

You use the following methods on the `QuerySelectColumns` class to specify columns in a type-safe way for row queries and ordering methods on result objects. All the methods return objects of type `IQuerySelectColumn`.

The column selection methods that end with `WithAlias` have a `String` as the first parameter. The value that you pass is an alias for the database column, which becomes the column name in the result of the row query.

The column selection methods that do not end with `WithAlias` have no parameter for column alias in the row query result. To access the column, you must use a numeric position value that starts from 0 or the name of the column as a `String` of format *`TABLE.COLUMN_NAME`*.

Method	Parameter	Description

`pathWithAlias`
`path`
		Creates a path to a column for ordering query results or to use in a row query. For example, the following lines select columns for a row query and order the results. You use the same syntax for ordering the results of an entity query.
		`var query = Query.make(Person)`
		`var results = query.select({`
		`  QuerySelectColumns.pathWithAlias("FName", Paths.make(Person#FirstName)),`
		`  QuerySelectColumns.path( Paths.make(Person#LastName))`
		`}).orderBy(QuerySelectColumns.path(Paths.make(Person#Las tName)))`

	`alias` : `String`	An alias for the column name in the row query result. This parameter has no purpose for a column that you use to order the query results.
	`path` : `PersistentPath`	A path expression that references the column to include in the row query result or to order the rows in the result set.

`dbFunctionWithAlias`
`dbFunction`
		Creates an SQL expression for ordering query results or to use as a column in a row query. For example, the following lines select columns for a row query and order the results. You use the same syntax for ordering the results of an entity query.
		`var query = Query.make(Address)`
		`var results = query.select({`
		`  QuerySelectColumns.dbFunction(DBFunction.Expr({"length (", query.getColumnRef("Address.City"), ")"}))`
		`}).orderBy(QuerySelectColumns.dbFunction(DBFunction.Expr({"length(", query.getColumnRef("Address.City"), ")"})))`
		WARNING Using `dbFunction` can cause the database to perform a whole-table scan. Ensure that use of these methods does not cause an unacceptable delay to the user interface.

	`alias` : `String`	An alias for the column name in the row query result. This parameter has no purpose for an expression that you use to order the query results.
	`func` : `DBFunction`	A `DBFunction` expression that creates a column to include in the row query result or to order the rows in the result set.

Other methods to access columns in building a query include the following:

Method	Parameter	Description

getColumnRef		Creates a reference to an entity field in terms of the database column that stores its values. Returns a ColumnRef. This method does not support chaining. Use this method to create a column reference for a second property in a comparison predicate or for an argument to a DBFunction method.
		`var query = Query.make(Address)`
		`var rowResults = query.select({`
		`  QuerySelectColumns.pathWithAlias("Country", Paths.make(Address#Country)),`
		`  QuerySelectColumns.dbFunctionWithAlias("LatestAddress" ,`
		`DBFunction.Max(Paths.make(Address#CreateTime)))`
		`})`

	propertyName : String	A String of format "*Entity.Property*".

Column specification in query results

To access a column in the results of an entity query, you use dot notation to access any property in the entire entity graph. For example, the following code prints the display name of a Person entity instance and the name of the user who created that Person. The default property is the display name, DisplayName.

```
var query = Query.make(Person)
var results = query.select()
for (person in results) {
  print(person.DisplayName + " " + person.CreateUser.Contact)
}
```

To access a column in the results of a row query, you use the getColumn method to access any column in the row. The results of a row query do not have a default property. For example, the following code is equivalent to the previous code for an entity query. The code creates a row query and prints the name of each person in the database and the name of the user who created that person.

```
var query = Query.make(Person)
var results = query.select({
  QuerySelectColumns.pathWithAlias("FName", Paths.make(Person#FirstName)),
  QuerySelectColumns.path(Paths.make(Person#LastName)),
  QuerySelectColumns.path(Paths.make(Person#CreateUser, User#Contact, Person#FirstName)),
  QuerySelectColumns.path(Paths.make(Person#CreateUser, User#Contact, Person#LastName))
})
for (person in results) {
  print(person.getColumn("FName") + " " + person.getColumn("Person.LastName") + " "
      + person.getColumn(2) + " " + person.getColumn(3))
}
```

The method that you use to retrieve the value of a column from a row in the result of a row query is the following. Use one of the parameters in the table. The return value of the method is an Object.

Method	Parameter	Description
getColumn		Retrieves the value of a column from a row in the result of a row query.
	alias : String	An alias for the column name in the row query result. Use this parameter if you defined the column by using pathWithAlias or dbFunctionWithAlias.
	propertyName : String	A String of format "*Entity.Property*". Use this parameter if you defined the column by using path or dbFunction.
	position : int	A zero-based int that is the numeric position of the column in the argument to the select method that created the results.

Predicate methods reference

The following table lists the types of comparisons and matches you can make with methods on the query object.

Predicate method	Parameter (Type)	Description
`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.

`between`	• Column name (`String`)
	• Start value (`Object`)
	• End value (`Object`)
		Checks whether a value is between two values. This method supports `String` values, date values, and number values.
		`query.between(Activity#PublicID, "abc:01",`
		`"abc:99")`
		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.

`compare`	• Column name (`String`)
	• Operation type (`Relop`)
	• Value (`Object`)
		Compares a column to a value. For the operation type, pass one of the following values to represent the operation type:
		• `Equals` – Matches if the values are equal
		• `NotEquals` – Matches if the values are not equal
		• `LessThan` – Matches if the row's value for that column is less than the value passed to the `compare` method.
		• `LessThanOrEquals` – Matches if the row's value for that column is less than or equal to the value passed to the `compare` method.
		• `GreaterThan` – Matches if the row's value for that column is greater than the value passed to the `compare` method.
		• `GreaterThanOrEquals` – Matches if the row's value for that column is greater than or equal to the value passed to the `compare` method.
		Pass these values without quote symbols around them. These names are values in the `Relop` enumeration.
		For the value object, you can use numeric types, String types, PolicyCenter entities, keys, or typekeys. For `String` values, the comparison is case-sensitive.
		Example of a simple equality comparison:
		`query.compare(Activity#Priority, Equals, 5)`
		Example of a simple less than or equal to comparison:
		`query.compare(Activity#Priority,`
		`LessThanOrEquals, 5)`
		To compare the value to the value in another column, generate a column reference and pass that instead.
		You can use algebraic functions that evaluate to an expression that can be evaluated at run time to be the appropriate type. For example:
		`var prefix = "abc:"  // string variable...`
		`var recordNumber = "1234"`
		`query.compare(Activity#PublicID, Equals,`
		`prefix + recordNumber)`
		Or combine a column reference and algebraic functions:
		`query.compare(Activity#Priority, Equals,`
		`DBFunction.Expr({`
		`query.getColumnRef("OldPriority"),`
		`"+", "10"}))`

Predicate method	Parameter (Type)	Description

`compareIgnoreCase`	• Column name (`String`)
	• Operation type (`Relop`)
	• Value (`Object`)
		Compares a character column to a character value while ignoring uppercase and lowercase variations. For example, if the following comparison succeeds:
		`query.compare("Name", Equals, "Acme")`
		Both of the following comparisons also succeed:
		`query.compareIgnoreCase("Name", Equals, Acme")`
		`query.compareIgnoreCase(Company#Name, Equals, "ACME")`

`compareIn`	• Column name (`String`) • List of values that could match the database row for the column (`Object[]`)	Compares the value for this column for each row to a list of non-null objects that you specify. If the column value for a row matches any of them, the query successfully matches that row. For example: `query.compareIn(Activity#PublicID, {"default_data:1", default_data:3"})`
`compareNotIn`	• Column name (`String`) • List of values that could match the database row for that column (`Object[]`)	Compares the value for this column for each row to a list of non-null objects that you specify. If the column value for a row matches none of them, the query successfully matches that row. For example: `query.compareNotIn(Activity#PublicID, {"default_data:1", default_data:3"})`
`contains`	• Column name (`String`) • Contains value (`String`) • Ignore case (`Boolean`)	Checks whether the value in that column for each row contains a specific substring. For example, if the substring is `"jo"`, it will match the value `"anjoy"` and `"job"` but not the values `"yo"` or `"ji"`. If you pass `true` to the final argument, Gosu ignores case differences in its comparison. For example: `query.contains(Person#FirstName, "jo", true /* ignore case */)` Test the use of the `contains` method in a realistic environment. Using the `contains` method as the most restrictive predicate on a query causes a full-table scan in the database because the query cannot use an index. **WARNING** For a query on a large table, using `contains` as the most restrictive predicate can cause an unacceptable delay to the user interface.
`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.
`startsWith`	• Column name (`String`) • Substring value (`String`) • Ignore case (`Boolean`)	Checks whether the value in that column for each row starts with a specific substring. For example, if the substring is `"jo"`, it will match the value `"john"` and `"joke"` but not the values `"j"` or `"jar"`. If you pass `true` to the Boolean argument (the third argument), Gosu ignores case differences in its comparison. For example: `query.startsWith(Person#FirstName, "jo", true /* ignore case */)` Note: If you choose case-insensitive partial comparisons, Gosu generates an SQL function that depends on your PolicyCenter and database configuration 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. **IMPORTANT** Test the use of the `startsWith` method in a realistic environment. Using the `startsWith` method as the most restrictive predicate on a query can cause a delay on the user interface.

Predicate method	Parameter (Type)	Description
`subselect`	The arguments provide set inclusion and exclusion predicates.	Perform a join with another table and select a subset of the data by combining tables.
`withinDistance`	• Identifier for column of type `SpatialPoint` (`IQueryablePropertyInfo`) • Path to location or spatial property. Use only in complex entity queries, such as those containing a join. (`String`) • Center (`SpatialPoint`) • Distance (`Number`) • Unit of distance OfDistance (`UnitOfDistance`)	Checks whether a location or spatial property on a column identifies a location that is within a given number of distance units of a center or reference location. For examples: • `addressQuery.withinDistance( Address.SPATIALPOINT_PROP.get(), SAN_MATEO, 10, UnitOfDistance.TC_MILE)` • `addressTable.withinDistance( Address.SPATIALPOINT_PROP.get(), "Person.PrimaryAddress.SpatialPoint", SAN_MATEO, 10, UnitOfDistance.TC_MILE)`

See also

- “Combining predicates with AND and OR logic ” on page 330
- “Comparing column values with each other” on page 326
- “Using set inclusion and exclusion predicates” on page 326

Predicate methods that support DBFunction arguments

The predicate methods that support a `DBFunction` as the comparison value are:

- `compare`
- `compareIgnoreCase`
- `between`
- `startsWith`
- `contains`
- `subselect`

Aggregate functions in the query builder APIs

The following methods on the `DBFunction` class support aggregate queries in the query builder APIs.

Method	Description	Example
`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")))`
`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")))`

Utility methods in the query builder APIs

The following methods on the `DBFunction` class provide useful functions in query builder code.

Method	Description	Example
`Constant`	Coerces a Gosu literal to an SQL literal in the generated SQL query that PolicyCenter submits to the database	`query.compare(Address#City, Equals, DBFunction.Constant("Los Angeles"))`
`DateDiff`	Returns the Interval between two date and time columns	`query.compare(DBFunction.DateDIff(DAYS, getColumnRef("AssignmentDate"), getColumnRef("EndDate")), LessThan, 15)`
`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)`
`DateFromTimestamp`	Returns the date portion of a timestamp	`query.compare(DBFunction.DateFromTimestamp( query.getColumnRef("CreateTime")), Equals, gw.api.util.DateUtil.currentDate())`
`Distance`	Returns the distance between the location of an entity instance and a reference location	`QuerySelectColumns.dbFunctionWithAliasDistance", DBFunction.Distance(addressQuery, "Address.SpatialPoint"))`
`Expr`	Returns a function defined by a list of column references and character sequences	

See also

- “Comparing the interval between two date and time fields” on page 322
- “Comparing parts of a date and time field” on page 323
- “Comparing the date part of a date and time field” on page 323
- “Comparing a column value with a literal value” on page 327
- “Creating and using an SQL database function” on page 352

chapter 24

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 not explicitly commit entity data manually.

When to use database transaction APIs

PolicyCenter groups all related entity instance changes together in a group called a *bundle*. Gosu provides APIs to manipulate entity data in bundles. These APIs also provide the ability to change how to save business data object modifications to the database. Saving to the database all related changes to current objects is called *committing* the bundle.

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 commits the bundle.

If errors prevent the entire commit action from safely completing, Gosu commits no data changes to the database. In certain cases, after an error occurs, Gosu retains the bundle. Depending on application context, the bundle is preserved, for example to let a user fix validation errors. In contrast, in a web service implementation, a commit failure typically results in discarding the temporary bundle and returning an error to the web service client.

The most important types of database transaction APIs provide the functionality to:

- Add an entity instance to a bundle
- Run code in a new bundle
- Commit a bundle explicitly, if necessary

WARNING Using database transaction APIs has significant 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 `runWithNewBundle`?	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 the Note column.	No	**IMPORTANT** It is dangerous and unsupported for rule set code to explicitly commit any bundle. Some rule sets, such as validation rules or pre-setup rules, have a main object that is already in the current writable bundle. If you change only that main entity and its subobjects, your Gosu rule set code does not typically require special database transactions.
PCF code in: • List views • View screens • Edit screens	No	Yes, for database query results.	No	**IMPORTANT** In most PCF code, it is dangerous to create additional bundles or explicitly commit bundles.Typical PCF widgets handle database transactions automatically. For example, after you enter Edit mode, the application creates a new bundle. After you click the **Save** button, the application commits the bundle with any data updates. In general, use the default PCF behaviors. In wizard user interface flow, you can optionally configure steps to commit the latest changes. Even in those cases your Gosu code does not explicitly commit the bundle.
PCF pages with asynchronous actions	Yes	Yes, for database query results.	No. The `runWithNewBundle` API method commits the bundle.	In unusual cases in PCF files, you might need more advanced bundle management. For example, rare self-contained code must change the database independent of success of the primary user action. Use the `runWithNewBundle` API method to create a new bundle for your entity data changes.
Plugin implementation code	No, in most cases. Plugin method arguments that are intended to be modified are in a writable current bundle. Startable plugins require creating a new bundle. Message reply plugins use a different mechanism for using bundles.	Yes, for database query results. Plugin method arguments that are intended to be modified are in a current bundle that is writable.	No, in most cases. For startable plugins, the `runWithNewBundle` API method commits the bundle.	If the application expects changes to plugin method arguments, the application sets up a current bundle.

Code context	Create a new bundle with `runWithNewBundle`?	Add entity instances to bundles before changing data?	Commit bundle explicitly for data changes?	Notes
Workflow code	No.	Yes, for database query results	No	The bundle for workflow actions is the bundle that the application uses to load the `Workflow` entity instance.
Custom batch processes	Yes	Yes, for database query results	No. The `runWithNewBundle` API method commits the bundle.	Use the `runWithNewBundle` API method to create a new bundle for your entity data changes.
Custom work queues	Yes	Yes, for database query results	No. The `runWithNewBundle` API method commits the bundle.	Use the `runWithNewBundle` API method to create a new bundle for your entity data changes. **WARNING** Custom work item code has access to a current bundle. However, using this bundle for your own business data changes is unsupported.
Web services	Yes, if the operation modifies data.	Yes, for database query results	No. The `runWithNewBundle` API method commits the bundle.	Use the `runWithNewBundle` API method to create a new bundle for your entity data changes.

See also

- “Adding entity instances to bundles” on page 390
- “Committing a bundle explicitly in very rare cases” on page 393
- “Running code in an entirely new bundle” on page 398

Overview of bundles

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.

A bundle is not thread-safe. Be aware of any concurrency issues when accessing entity instances in a bundle or committing those instances to the database. If multiple users can access the entity instances in a bundle simultaneously, you must use external synchronization to ensure reading and writing correct data, such as for property values.

WARNING Use external synchronization to ensure data integrity for concurrent access to entity instances.

The two basic types of bundles are read-only bundles and writable bundles. A database query places the results of the query are in a temporary read-only bundle. To change any data, you must copy the contents of a read-only bundle to a writable bundle.

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 that has uncommitted changes, no entity data in the database changes.

WARNING The base configuration contains visible usages of the entity instance method `setFieldValue`. The `setFieldValue` method is reserved for Guidewire internal use. Calling this method can cause serious data corruption or errors in application logic that may not be immediately apparent. You must obtain explicit prior approval from Guidewire Support to call `setFieldValue` as part of a specific approved use.

See also

- "Making an entity instance writable by adding to a bundle" on page 391
- "Adding entity instances to bundles" on page 390
- "Getting the bundle of an existing entity instance" on page 392
- "Getting an entity instance from a public ID or a key" on page 392
- "Creating new entity instances in specific bundles" on page 393
- "Committing a bundle explicitly in very rare cases" on page 393
- "Determining what data changed in a bundle" on page 395
- "Concurrency" on page 417

Accessing the current bundle

In most programming contexts, the application has already prepared a current bundle, 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()
```

In batch processes and WS-I web service implementations there is no current bundle. You need to create a new bundle for your data changes.

Warning about transaction class confusion

PolicyCenter provides more than one `Transaction` type in Gosu. This topic is about the class in the `gw.transaction` package, `gw.transaction.Transaction`. Do not confuse the `gw.transaction.Transaction` class with the `Transaction` entity type or type list.

If you use the `gw.transaction.Transaction` API, you can use a Gosu `uses` statement such as the following:

```
uses gw.transaction.Transaction
```

Adding entity instances to bundles

A set of entity instances that a query retrieve from the database is not writable. You must add an entity instance to a writable bundle to make the instance writable.

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. 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.

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, and then set its value to 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 the result 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 from the database. Take care to avoid dangling references to the deleted object if you perform this action:

```
bundle.delete(obj)
```

In most programming contexts, it is 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 PolicyCenter perform its default bundle commit behavior.

See also

- “Overview of the query builder APIs” on page 311
- “When to use database transaction APIs” on page 387
- “Committing a bundle explicitly in very rare cases” on page 393
- “Removing entity instances from the database” on page 394
- “Running code in an entirely new bundle” on page 398

Moving a writable entity instance to a new writable bundle

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 exception, be careful not to modify the entity in more than one bundle.

Be aware that even if the entity instance is unmodified at the time you copy it to the new bundle, problems can occur 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 failed commit attempt 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, such as when switching from view mode to edit mode. Refreshing an entity is an internal process that discards a cached version and gets the latest version from the database. This process reduces the likelihood of concurrent data change exceptions. Gosu does not provide public API to refresh entity instances.

See also

- "Making an entity instance writable by adding to a bundle" on page 391

Getting the bundle of an existing entity instance

If you have a reference to an entity instance in a bundle, you can use the *object*`.Bundle` property to get a reference to its bundle. Use this property if you want entity changes to happen with existing changes that you know are in a writable bundle.

For example, you can use the bundle reference and call its `add` method to include additional entity instances. When PolicyCenter commits that bundle, all changes to the database happen in one database transaction.

See also

- "Making an entity instance writable by adding to a bundle" on page 391

Getting an entity instance from a public ID or a key

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 `PublicID` property of entity instance. Integration code frequently needs to manipulate objects by public ID.

The object key

A key represents the internal `Id` property entity instance, 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 by using a constructor that takes the entity type and the numeric ID. For example:

```
var myAddress = gw.transaction.Transaction.getCurrent().loadBean(new Key(Address, 123))
```

To load an entity by a public ID, use the query builder API. For example:

```
Query.make(Policy).compare(Policy#PublicID, Relop.Equals, myPublicId).select().AtMostOneRow
```

If you use the query builder API, the entity reference returned is in a read-only bundle. If you need to modify the data and save changes to the database, you must first add the entity instance to a writable bundle.

See also

- "Adding entity instances to bundles" on page 390
- "Query builder APIs" on page 311

Creating new entity instances in specific bundles

If you pass arguments to the `new` operator, Gosu calls a specific constructor defined for that Java type. If there are multiple constructors, Gosu uses the number and types of the arguments that you pass to choose the correct constructor. For Guidewire business entities such as `Policy` and `User`, typical code passes no arguments to create the entity. If you pass no arguments, Gosu creates the entity in the current bundle. Using the current bundle 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 instance. Pass an existing entity instance to create the new entity instance in the same bundle. Pass a bundle to explicitly use for the new entity instance. The following table compares different arguments to the `new` operator.

Arguments to `new` operator	Example	Result
No arguments	`new Note()`	Constructs a new `Note` entity in the current bundle. Changes related to the current code submit to the database at the same time as 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. In some programming contexts, such as batch processes and WS-I web service implementations, there is no automatic current bundle.
An entity instance	`new Note(myPolicy)`	Constructs a new `Note` entity in the same bundle as a given `Policy` entity instance. Changes to the `Policy` are committed to the database at the same time as the new note, and all other changes in the bundle.
A bundle	`new Note(myPolicy.bundle)`	Same as previous table row

See also

- "Using the operator new in object expressions" on page 86
- "Running code in an entirely new bundle" on page 398

Committing a bundle explicitly in very rare cases

Committing a bundle sends all changes for entity instances in this group of entities to the database. A successful commit can perform any combination of adding, changing, or removing entity instances from the database. In typical code, do not explicitly commit a bundle.

For data changes, typically 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. There is no reason to explicitly commit this bundle.
- In some cases, you must create an entirely new bundle. Do not explicitly commit the bundle. The `runWithNewBundle` API commits the bundle automatically when your code completes.

  WARNING Only commit a bundle if you are sure it is appropriate for that programming context. Otherwise, you could causes 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.

For web service implementations that need to change data, use the `runWithNewBundle` API method. The `runWithNewBundle` API method commits the bundle for you when your code completes.

If you are sure that committing the bundle is appropriate, 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, use the *entity*.bundle property to get its bundle. Committing a bundle commits everything in the bundle, not just the instance from which you got the bundle reference. For example:

```
var bundle = myPolicy.bundle
bundle.commit() // Commit all entities in the bundle
```

See also

- "When to use database transaction APIs" on page 387
- "Running code in an entirely new bundle" on page 398

Removing entity instances from the database

WARNING Guidewire strongly recommends never to delete entities from the database. An entity instance can include foreign key links to other entity instances and other entity instances can link to the deleted entity instance. Removal of an entity instance can compromise data integrity. If you must delete an entity instance, using the remove method on the entity instance is strongly preferable to using the Bundle interface's delete method.

Entity instance remove method

Mark an entity instance for removal from the database by using the remove method on the entity.

When the bundle is committed, the action that PolicyCenter takes depends on whether the entity is retireable. If the entity's data model configuration does not specify that the entity is Retireable, the entity instance is deleted from the database. If the data model specifies that the entity is Retireable, PolicyCenter retires the entity instance rather than deleting it. Additional actions of the remove method depend on the entity type. Typically, remove deletes owned child entity instances to which arrays in that instance refer even if those entity types are Retireable. Some entity types perform additional clean-up actions as part of their remove implementation.

For example:

```
uses gw.transaction.Transaction

// Make a query of Address instances
var query = Query.make(Address)

// Query for addresses created today
query.compare(
  DBFunction.DateFromTimestamp(query.getColumnRef("CreateTime")), Equals, DateUtil.currentDate())

for (address in query.select()) {
  address = bundle.add(address)
  address.remove()
}
```

Bundle interface delete method

The Bundle interface's delete method also provides the functionality to delete an entity instance from the database. This method does not perform any additional actions to ensure database integrity. The remove method on some entity types does perform additional clean-up. For this reason, if you must delete an entity instance, using the remove method on the entity instance is strongly preferable to using the Bundle interface's delete method.

When the bundle is committed, the entity instance is deleted from the database. If the entity's data model configuration declares the entity as `Retireable`, the entity is not deleted, but is retired instead. The `delete` method removes only the specified entity instance from the database, not linked entity instances.

See also

- *Configuration Guide*

Determining what data changed in a bundle

In rule set code, you sometimes need to identify data that 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 `getOriginalValue`	Behavior of `isFieldChanged`
Scalar value	Returns the original simple value, such as a `String` value like `"John Smith"`, a number like 234, or any other non-entity and non-array type.	Returns `true` if and only if the actual property value changed to a different actual value. If the value changed to a different value and then back to the original value, `isFieldChanged` would return `false`.

Entity	Guidewire applications represent links to entity subobjects as a foreign key called the *internal ID*. This foreign key is the *entity*`.Id` property on the destination entity. Note that the `Id` property is different than the `PublicID` property. If you call `getOriginalValue` with a foreign key field, you get an internal ID as a `Key` object. A `Key` object encapsulates the entity type and the unique ID value for that object.
	To get a reference to the entity instance with the original ID use the bundle method `loadByKey`. For example:
	`var k = e.getOriginalValue("Address1")`
	`var bundle = Transaction.getCurrent()`
	`var addy = bundle.loadByKey(k) as Address`
	Remember to call the `getOriginalValue` method on the correct entity instance.
		Returns `true` if and only if the foreign key `Id` value for the subobject is the same in the original entity. Remember to call the `isFieldChanged` on the correct entity instance.
		Keep in mind that checks for a changed entity could return `false` even if there are changes on properties of subobjects.
		Be aware that the foreign key is a link to the `Id` property of the target object. The `PublicID` property of the target object is not examined.

Array	Returns an array containing the set of array elements that were originally persisted. 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 by calling `getOriginalValue` on each array element for each property.	Returns `true` if any elements were added or removed to the array. Additionally, if the array is defined in the data model configuration as an owned array, if any properties on an array element changed, `isFieldChanged` returns true. This does not apply to non-owned arrays. **Note:** Because entity array order is not meaningful, the order is not checked by `isFieldChanged`. In this case, `isFieldChanged` returns the same value as the array method `isArrayElementAddedOrRemoved`.

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.

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
```

See also

- “Getting an entity instance from a public ID or a key” on page 392

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. This property returns a Boolean value of `true` if the entity instance 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 the `ChangedFields` property. That property value has the type `java.util.Set`.

The following example uses a Gosu block that iterates across the set of changed properties:

```
if (myAddress.Changed) {
  myAddress.ChangedFields.each( \ e -> print("Address has changed property: " + e))
}
```

See also

- "What are blocks?" on page 175
- "Collections" on page 183

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. You use methods to compare the array on the persisted entity instance to the array on the cached version in the bundle.

The following methods are available only for properties directly defined on an entity in the data model configuration files as entity arrays. These methods do not work on other types of properties or methods. For example:

- These methods do not work with Gosu enhancement properties that return entity arrays.
- These methods do not work with other virtual properties such as those implemented in internal Java code.

Use the *Data Dictionary* to check whether a property is a database-backed property or a virtual property.

Entity methods for getting changes to array properties

The following table lists the methods on an entity instance that you can use to get entity array changes.

Entity method	Argument	Return type	Returns
`getAddedArrayElements`	Array property name	Entity array	A list of array elements that were added. If there are none, returns an empty array.
`getChangedArrayElements`	Array property name	Entity array	A list of array elements that were changed. If there are none, returns an empty array.
`getRemovedArrayElements`	Array property name	Entity array	A list of array elements that are marked for deletion if the bundle commits. If there are none, returns an empty array.
`isArrayElementChanged`	Array property name	`boolean`	`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`	`true` if and only if any array elements were added or removed.

Getting added, changed, or deleted entities in a bundle

A bundle's set of entity instances might include multiple types of entities. You can get the set of changes by using properties on the bundle object.

Bundle methods for getting added, changed, or deleted entities

The following table lists the methods that you can use to find bundle changes.

Type of entities to return	Bundle API method	Description
New entity instances	`bundle.InsertedBeans`	Entity instances in this bundle that are entirely new objects, which have never been committed to the database, appear in this set. This bundle property returns an iterator that iterates across the set of entity instances.

Type of entities to return	Bundle API method	Description
Changed entity instances	`bundle.ChangedBeans`	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.
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 retired. This bundle property returns an iterator that iterates across the set of entity instances.
Unmodified entity instances in the current bundle	Not available	An entity might be added to a writable bundle but not yet modified. There is no API method 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, PolicyCenter 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 need to run Gosu code in an entirely new transaction, so you create a new bundle. 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.

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 by using the `bundle.add(`*`entity`*`)` method. Remember to save the return result of the `add` method.

Gosu immediately runs this code synchronously. If the block succeeds with no exceptions, `runWithNewBundle` commits the new bundle after your code completes. For the most concise and readable 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 discards the bundle and does not commit the changes.

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 in 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.

WARNING Only commit a bundle if you are sure that it is appropriate for that programming context. Otherwise, you could causes 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))
}

...
```

This apparently simple block is not self-contained because the code uses `customerID`, which is the argument to the `myAction` method and is declared outside the scope of the block. The ability of a block to use variables from its creation context is called *variable capturing*.

In some cases, you need to return information from your block. You can return information by more advanced use of variable capturing. The following example demonstrates variable capturing by creating a local variable in the outer scope (`myAction`) and then setting its value from within the block. The outer scope can use 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" are 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
}
```

See also

- “Blocks” on page 175
- “Variable scope and capturing variables in blocks” on page 177
- “Adding entity instances to bundles” on page 390
- “Getting changes to entity arrays in the current bundle” on page 397

Creating a bundle for a specific PolicyCenter user

The `gw.transaction.Transaction` class provides a signature of the `runWithNewBundle` method to associate a specific user with the newly created bundle. You use this method signature in contexts in which there is no built-in user context, such in the startable plugins feature. The method signature is:

```
gw.transaction.Transaction.runWithNewBundle(\ bundle -> YOUR_BLOCK_BODY, user)
```

For the second argument to the method, you can pass either a `User` entity instance or the user name as a `String`.

Guidewire strongly recommends that you do not use the Guidewire sys user (System User), or any other default user, to create a new bundle.

Effects of exception handling on database commits

Typically, if your code throws an exception, Gosu does not commit the current database transaction. This behavior applies to rule sets, PCF code, and other contexts that have a current bundle.

Throw an exception from your code to get this behavior. If you catch exceptions in your code, to ensure that the current bundle is not committed in the current action, rethrow the exception.

Bundles and published web services

Web service implementation classes must explicitly manage all bundles, for both reading and writing entity data. Because web services do not support entity instances as arguments or return types, the interaction between web service implementation classes and bundles is straightforward.

Entity cache versioning, locking, and entity refreshing

The application caches entity data for faster access. The `gw.transaction.Transaction` API methods that refer to original entity instances check against the database version 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. The application may have loaded the database row long before the recent application change. Although the database data may have changed since the time of loading, some safeguards prevent concurrent data access in most cases. For typical entity access, the server prevents committing an entity instance if the instance changed in the database after the entity loaded in the bundle and before the bundle commits.

Entity instance versioning and the entity touch API

PolicyCenter 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, PolicyCenter checks this version number property in each entity instance with the latest version in the database. PolicyCenter 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. PolicyCenter 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. Your code reads the values and changes most or all of the objects in one database

transaction. In one case, the code makes changes to only B, C, and D. There remains one object (A) for which you did not change properties. Therefore, A is not in the same bundle as B, C, and D. In this case, neither does the database row for A change nor does its version number increment, which will not be a problem in some cases.

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 PolicyCenter 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 preupdate rules to run for object A.

To force PolicyCenter to increment the entity version number (*obj*`.BeanVersion`), update the modified time, and force preupdate 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 a commit. The server throws concurrent data change exceptions if two different threads or two different servers in a cluster simultaneously modify the same entity at the same time.

Concurrent data change exceptions always occur if two application users attempt to change the same record from the application user interface.

If all of the following conditions are true, a concurrent data change exception does not occur:

- A batch process or messaging plugin made the update that conflicts with a change from the user interface.
- The user that made the update is a `system` user. In the base configuration, in the `User` record of a system user, the `SystemUserType` property is `sysadmin` or `sysservices`.
- The `BeanVersion` field in the database record differs only by 1. In this case, only the batch process or messaging plugin has updated the record.
- The changes that the concurrent change makes are not in fields that the batch process or messaging plugin updated.

User interface bundle refreshes

In some cases, the PolicyCenter user interface internally refreshes bundle entity data with the latest version from the database. For example, PolicyCenter refreshes the data when changing from view-only to edit mode.

Gosu does not provide a public API for you to programmatically refresh a bundle's entity data.

Details of bundle commit steps

PolicyCenter commits a bundle to save data to the database by performing the following steps.

Note: Bundle commit does not perform the following preupdate rule sets and validation rule sets: archiving, geocoding changes, messaging reply, messaging preprocessor, upgrade actions, and XML import, including administrative data import. For this reason, there are no messaging events nor concurrent data change exceptions for these actions. These actions always overwrite an entity instance in the database with the instance in the bundle.

1. PolicyCenter reserves a connection from the connection pool.
2. Internally, PolicyCenter saves the state of the bundle. PolicyCenter uses this snapshot to revert to the current state of data if writing to the database fails.
3. PolicyCenter runs preupdate rule sets.
4. PolicyCenter refreshes all entity instances that already existed but are unchanged in the bundle. This action ensures that validation rules, which run next, get the newest versions of those entity instances.

5. PolicyCenter triggers validation rule sets.
6. PolicyCenter sets properties that exist on editable and versionable entities: `updateTime`, `createTime`, and `user`.
7. PolicyCenter increments the version number on each modified entity instance.
8. Any entity instances that are new (not yet in the database) have a temporary internal ID. An internal ID is the `entity.Id` property. PolicyCenter creates the permanent internal ID for each new entity instance and assigns the value to the `entity.Id` property. PolicyCenter sets any foreign key references to refer to the new internal ID value rather than the temporary ID.
9. PolicyCenter computes the set of new, changed, and removed entity instances in the bundle.
10. PolicyCenter 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, PolicyCenter triggers Event Fired rule sets once for each messaging destination. For example, *`ENTITYNAME`*`Added`, *`ENTITYNAME`*`Changed`, or *`ENTITYNAME`*`Removed` events. If more than one destination listens for the same event, the Event Fired rule set executes once for each messaging destination.
12. PolicyCenter writes all changed entities to the database connection. During this step, PolicyCenter checks for concurrent data change exceptions.
13. PolicyCenter attempts to commit the database connection. This action either completely succeeds or fails.
14. On success, PolicyCenter 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 a PolicyCenter cluster.

    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. After a server receives this signal, entity instances in a remote entity cache immediately show the data as requiring an update, but do not immediately reload data 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 also

- “Entity instance versioning and the entity touch API” on page 400
- *Integration Guide*
- *System Administration Guide*

chapter 25

Type system

Gosu provides several ways to gather information about an object or other type. Use this information to debug or to determine program behavior at run time.

Basic type checking

Every Gosu object and value has a type. For example, the type of an instance of a class is the class itself.
Gosu provides the following operators to retrieve or test an item's type.

- The `typeof` operator retrieves the run-time type of an item.
- The `typeis` operator tests the type of an item.

Retrieve type with typeof

Use the `typeof` operator to retrieve the run-time type of a data item.
The syntax of the `typeof` expression is shown below.

```
typeof expression
```

The example below demonstrates the syntax. The sample `typeof` expression evaluates to `true`.

```
var myVar : Integer = 10
if (typeof myVar == Integer) print("true")   // typeof myVar == Integer
```

Comparing the retrieved type to a subtype of the retrieved type evaluates to `false`. For example, the `Integer` type is a subclass of the `Number` class, which itself is a subclass of the `Object` class. All of the `typeof` expressions shown below evaluate to `false`.

```
if (typeof myVar == Number) print("false")   // typeof myVar != Number
if (typeof myVar == Object) print("false")   // typeof myVar != Object
```

The type of an entity or typekey is a parameterized type of `Type<t>`.
The following example shows some `Type<t>` types of a `Producer` entity and a `NoteSecurityType` typekey. The `typeof` expressions shown below evaluate to `true`.

```
if (typeof Producer == Type<Producer>) print("true")
if (typeof Producer == Type<entity.Producer>) print("true")
if (typeof NoteSecurityType == Type<NoteSecurityType>) print("true")
if (typeof NoteSecurityType == Type<typekey.NoteSecurityType>) print("true")
```

The `Type<t>` syntax must be used in comparison expressions of entities or typekeys that use the `typeof` operator. The expressions shown below evaluate to `false` because they do not use the required `Type<t>` syntax.

```
if (typeof Producer == Producer) print("false")
if (typeof Producer == entity.Producer) print("false")
if (typeof NoteSecurityType == NoteSecurityType) print("false")
if (typeof NoteSecurityType == typekey.NoteSecurityType) print("false")
```

The example below demonstrates how the run-time type retrieved by the `typeof` operator can differ from the type specified when the variable was defined.

```
var myVar : Object = 10
print(typeof myVar)         // Run-time type is java.lang.Integer
```

The run-time type of a `null` value is the `void` type as demonstrated below.

```
var myVar : Boolean = null
print(typeof myVar)         // Run-time type of the null value is void
```

Test type with typeis

A `typeis` expression tests the type of an object. If the object is of either the tested type or a subtype of the tested type, the expression evaluates to `true`.

The syntax of a `typeis` expression is shown below.

```
OBJECTtypeis TYPE
```

The definition of a `myVar` variable of type `Integer` is shown below. The `Integer` type is a subclass of the `Number` class, which itself is a subclass of the `Object` class. Thus, all the `typeis` expressions shown below evaluate to `true`.

```
var myVar : Integer = 10

if (myVar typeis Integer) { print("true") }   // true because myVar is of type Integer
if (myVar typeis Number) { print("true") }    // true because Number is a subtype of Integer
if (myVar typeis Object) { print("true") }    // true because Object is a subtype of Integer
```

The type of an entity or typekey is a parameterized type of `Type<t>`.

The following example shows some `Type<t>` types of a `Producer` entity. The `Producer` class is a subclass of the `BeanBase` class. All of the `typeis` expressions shown below evaluate to `true`.

```
if (Producer typeis Type<Producer>) { print("true") }
if (Producer typeis Type<entity.Producer>) { print("true") }
if (Producer typeis Type<BeanBase>) { print("true") }   // true because BeanBase is a subtype of Producer
```

The following example shows some `Type<t>` types of a `NoteSecurityType` typekey. The `NoteSecurityType` typekey is a subclass of the `Object` class. All of the `typeis` expressions shown below evaluate to `true`.

```
if (NoteSecurityType typeis Type<NoteSecurityType>) { print("true") }
if (NoteSecurityType typeis Type<typekey.NoteSecurityType>) { print("true") }
if (NoteSecurityType typeis Object) { print("true") }
```

Some methods, parameters, and return values use the interface type `IType` which is analogous to the parameterized `Type<t>`.

If the compiler determines that a particular `typeis` expression can never evaluate to `true` at run time then a compile error is generated. This behavior is demonstrated in the following example which generates a compile error because the `Boolean` variable can never hold an `Integer` value.

```
var myVar : Boolean

if (myVar typeis Integer) { print("Compile Error") }   // Compile Error, Boolean can never be Integer
```

Basic type coercion

Gosu provides coercion or casting syntax to cast an expression to a specific type. The syntax is shown below.

```
expression as TYPE
```

The expression must be compatible with the type. The following table shows the results of casting a simple numeric expression into one of the Gosu-supported data types. If you try to cast an expression to an inappropriate type, Gosu throws an exception.

Expression	Data type	Result or error
`(5 * 4) as Boolean`	`Boolean`	`true`
`(5 * 4) as String`	`String`	`"20"`
`(5 * 4) as Array`	Not applicable	`Type mismatch or possible invalid operator. java.lang.Object[] is not compatible with java.lang.Double`

The `as` keyword can be used to cast an array type to another compatible array type. For example, `String` is a subtype of `Object`, so an array of type `String` can be cast to an array of type `Object`.

```
var objArray : Object[]
objArray = new String[] {"a", "b"} as Object[]   // Okay to cast String[] to Object[]
```

The reverse operation of casting an `Object` to a `String` is invalid, as demonstrated below.

```
var objArray = new Object[1]
objArray[0] = "foo"
var strArray = objArray as String[]   // Run-time error attempting to cast Object[] to String[]
```

The compiler can detect and produce a compilation error when two referenced types are incompatible. For example, an `Integer[]` can never be converted to a `String[]`, so the following code statements produce a compilation error.

```
var strArray : String[]
strArray = new Integer[] {2, 3} as String[]   // Compilation error attempting to cast Integer[] to String[]
```

In some cases, you know that all the objects in an array are of a particular subtype of the defined type of the array. In other cases, you filter an array to retrieve only objects of a particular subtype. As shown in a preceding code example, you cannot use the `as` keyword to convert an array to an array of a subtype. Instead, you can use the Gosu enhancement method `cast` to make an array of the subtype. You can then perform operations on the new array that are specific to the subtype and not available to the parent type. The following code demonstrates how to use the `cast` method.

```
var objArray = new Object[1]
objArray[0] = "foo"
var strArray = objArray.cast(String)

var objArray2 = new Object[2]
objArray2[0] = "foo"
objArray2[1] = 42
var strArray2 = objArray2.where(\aid -> aid typeis String).cast(String)
```

You cannot use the `cast` method to convert incompatible types. The compiler does not detect this type of error. The following code produces a run-time error.

```
var strArray : String[]
strArray = new Integer[] {2, 3}.cast(String)   // Run-time error attempting to cast Integer[] to String[]
```

Automatic type downcasting

To improve the readability of Gosu code, Gosu automatically downcasts a type after a `typeis` or `typeof` expression if the type is a subtype of the specified type. Automatic type downcasting is particularly valuable for `if` and `switch` code blocks. Gosu confirms that the typed object has the more specific subtype within the code block. Accordingly, Gosu implicitly downcasts the variable's type to the relevant subtype for the block. There is no need to explicitly cast the type with an `as` *`TYPE`* statement.

A common pattern for this feature is shown below.

```
var VARIABLE_NAME : TYPE_NAME

// This code requires SUBTYPE_NAME to be a subtype of TYPE_NAME
if (VARIABLE_NAME typeis SUBTYPE_NAME) {
  // Use the VARIABLE_NAME variable as a SUBTYPE_NAME without explicit casting
  ...
}
```

For example, the following code shows a variable declared as an `Object`, but automatically downcast to `String` when referenced within the `if` code block.

```
var x : Object = "nice"
var strlen = 0

if ( x typeis String ) {
  strlen = x.length       // x automatically downcast to a String which has a length property
}
```

Note that `length` is a property on `String`, not `Object`. The automatic downcasting of the variable eliminates the need to explicitly cast it. The following code is equivalent to the previous code, but has an unnecessary explicit cast.

```
var x : Object = "nice"
var strlen = 0

if ( x typeis String ) {
  strlen = (x as String).length    // Do not need to explicitly cast x as a String
}
```

Best practice recommends utilizing automatic downcasting when it occurs and eliminating unnecessary explicit casting.

When automatic downcasting occurs

Automatic downcasting occurs when executing the following types of statements.

- `if` statements
- `switch` statements

```
uses java.util.Date

var x : Object = "neat"
switch ( typeof x ){
  case String :
    print( x.charAt( 0 ) )   // x is automatically downcast to type String
    break
  case Date :
    print( x.Time )          // x is automatically downcast to type Date
    break
}
```

- The `true` portion of ternary conditional expressions

```
x typeis String ? x.length : 0   // x is automatically downcast to type String
```

The automatic downcast occurs only for the `true` portion of the ternary expression. In the `false` portion, the variable reverts to its original type.

The automatic downcast is terminated and the object is interpreted as being of its original type when any of the following conditions occurs.

- Reaching the end of the code block's scope
  - The end of an `if` code block
  - The end of a `case` code block within a `switch` statement
  - The end of the `true` portion of a ternary conditional expression
- Assigning any value to the variable that you checked with `typeis` or `typeof`. This behavior applies only to `if` and `switch` statements.
- Assigning any value to any part of an entity path that you checked with `typeis` or `typeof`. This behavior applies only 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 not terminated by a `break` statement. The following `switch` statement demonstrates this rare behavior.

```
uses java.util.Date

var x : Object = "neat"
switch( typeof x ){
  case String :
    print( x.charAt( 0 ) )  // x is automatically downcast to type String
  case Date :               // Code execution falls through to next case
    print( x.Time )         // x reverts to original type of Object which does not have Time property
    break
}
```

When the `x` object of type `Object` attempts to access a non-existent `Time` property, a compile error occurs. To work around this behavior, explicitly cast the `x` object to a type `Date` as shown below.

```
print( (x as Date).Time )  // Explicitly cast x to type Date to access the Time property
```

Type metadata properties and methods

The type `Type` is the root type of all types. Every `Type` includes metadata properties and methods, such as the methods `getDisplayName` and `isAssignableFrom`.

General type metadata

General type metadata is available for all types of data. Some of the properties and methods available in general type metadata are described in the following table. For a complete list of properties and methods, refer to the "Gosu API Reference."

Property or method	Description
`getDisplayName`	Retrieves the human-readable name of the type.
`getSupertype`	Retrieves the supertype of the type. If no supertype exists, returns `null`.
`isAbstract`	If the type is abstract, returns `true`.
`isArray`	If the type is an array, returns `true`.
`isFinal`	If the type is final, returns `true`.

Property or method	Description
`isGenericType`	If the type is generic, returns `true`.
`isInterface`	If the type is an interface, returns `true`.
`isParameterizedType`	If the type is parameterized, returns `true`.
`isPrimitive`	If the type is primitive, returns `true`.
`isAssignableFrom`	If the type is assignable from a variable of a specified type, returns `true`. Boxed types, such as `Boolean` and `Integer`, can be assigned to and from other boxed types only. If either variable type referenced in the `isAssignableFrom` expression is a primitive, such as `boolean` or `int`, the method returns `false`.
`TypeInfo`	Contains metadata about properties and methods of this type.

Accessing general type metadata

To access general type metadata, retrieve the `Type` property of the data by using the `typeof` operator. The following example accesses various type metadata values on the `java.util.ArrayList` type.

```
var stringList = {"cat", "dog"}
var t = typeof stringList

print("Name = " + t.getDisplayName)
print("Supertype = " + t.getSupertype.getDisplayName())
print("isAssignableFrom(Object) = " + t.isAssignableFrom(Object))
print("Example method name = " + t.TypeInfo.Methods[0].getDisplayName())
```

Alternatively, the `Type` property of a data type can be referenced directly to access the type's metadata. The following example demonstrates the appropriate syntax to access the metadata of the `ArrayList` class.

```
print("Name = " + ArrayList.Type.getDisplayName())
print("Supertype = " + ArrayList.Type.getSupertype().getDisplayName())
print("isAssignableFrom(Object) = " + ArrayList.Type.isAssignableFrom(Object))
print("Example method name = " + ArrayList.Type.TypeInfo.Methods[0].getDisplayName())
```

The output from the example code is identical for both of the demonstrated access methods.

```
Name = java.util.ArrayList<java.lang.Object>
Supertype = java.util.AbstractList<java.lang.Object>
isAssignableFrom(Object) = false
Example method name = clone
```

Type metadata for entities and typekeys

Gosu typekeys and some Gosu entities have additional type metadata that is not included with the general metadata. For example, Gosu typekeys have a metadata method called `getTypeKeysByCategories` that is available only on typekey data types. To access the entity and typekey metadata, reference the data's `TYPE` property and `get` method. The code sample below demonstrates the appropriate syntax.

```
var myTypekeys = new TypeKey[10]
NoteSecurityType.TYPE.get().getTypeKeysByCategories(myTypekeys)
```

The `TYPE` property is available only on Gosu entity and typekey data types.

Primitive and boxed 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 to non-primitive versions or back again by the Gosu language in almost all cases. For example, `int` can be cast to `Integer` and `Boolean` cast to `boolean`. In typical code, you do not need to know the differences.

The `boolean` type is a Java primitive which is also called an "unboxed" type. In contrast, `Boolean` is a class that implements a "boxed" type of the `boolean` primitive. A boxed type is a primitive type wrapped in the shell of a class. Boxed types are useful for code that requires all values to have the common ancestor type `Object`. For example, Gosu collection data types typically require members to have types that descend from `Object`. Thus, collections can contain `Integer` and `Boolean`, but not the primitives `int` or `boolean`.

The Gosu boxed types use the Java primitive types. The boxed types are defined in the `java.lang` package, such as `java.lang.Integer`. Code execution performance is slightly improved when using a primitive rather than its boxed version.

There is an important difference between primitive and boxed types when handling uninitialized values. Variables declared of a primitive type cannot hold the `null` value. However, `null` can be assigned to variables of type `Object` and any subtype of `Object`.

Compound types

To implement other features, Gosu supports a special type called a compound type. A compound type combines an optional base class and additional interfaces. Typical usage of compound types occurs only when Gosu automatically creates a variable with a compound type because of type inference. In extremely rare cases, Gosu allows a variable to be declared explicitly with a compound type.

Suppose you use the following code to initialize list values with different types of objects. When used with initializers, the run-time type is always ArrayList<Object, Object>. The compile-time type depends on what you pass to the initializer. Gosu infers the type of the result list to be the least upper bound of the components of the list. 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 behavior ensures that 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:

- Zero classes or one class, which might be `java.lang.Object`, if no other common class is found
- One or more interfaces

At compile time, the list and its elements use the compound type. You can use the list and its elements with APIs that expect those interfaces or the ancestor element.

For example, the following code creates classes that extend a parent class and implement an interface.

```
interface HasHello {
  function hello()
}

class TestParent {
  function unusedMethod() {}
}

class TestA extends TestParent implements HasHello{
  function hello() {print("Hi A!")}
}

class TestB extends TestParent implements HasHello{
  function hello() {print("Hi B!")}
}

var newList = { new TestA(), new TestB()}

print("Run-time type = " + typeof newList)
```

This code prints the following line.

```
Run-time type = java.util.ArrayList<java.lang.Object>
```

The compile-time type is a combination of:

- The class `TestParent`, which is the most specific type in common for `TestA` and `TestB`
- The interface `HasHello`, which both `TestA` and `TestB` implement

Gosu also creates compound types in the special case of using the `delegate` keyword with multiple interfaces.

Using reflection

If 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 run time. For example, instead of calling an object method, you can get a list of methods from its type, or use a `String` run-time value to call a method by name. 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.

Only use reflection if there is no other way to do what you need. 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 problems at compile time. Detecting errors at compile time is better than encountering unexpected behavior at run time.

The following example shows two different approaches for getting the `Name` property from a type.

```
print(Integer.Type.Name) // Getting properties directly from a Type
print((typeof 29).Name)  // Getting the Type of an object
```

This code prints the following output.

```
java.lang.Integer
int
```

Getting properties by using reflection

The *`type`*`.TypeInfo` object includes a property called `Properties`, which contains a list of type properties. Each item in that list includes metadata properties such as for the name (`Name`) and a short description (`ShortDescription`). Because of performance implications and the inability to find errors at compile time, only use reflection if there is no other way to do what you need.

To examine how reflection provides information about class properties, paste the following code into the Gosu Scratchpad.

```
var object = "This is a string"
var propNames = ""
var props = (typeof object).TypeInfo.Properties

for (prop in props) {
  propNames = propNames + prop.Name + "  "
}

print(propNames)
```

This code prints lines similar to the following output.

```
Class  Bytes  Empty  CASE_INSENSITIVE_ORDER  Digits  AsBigDecimal  length  size  HasContent
NotBlank  Alpha  AlphaSpace  Alphanumeric  AlphanumericSpace  Numeric  NumericSpace  Whitespace
```

Array syntax for getting a property by using reflection

You can use reflection to access properties by using the square bracket syntax, similar to using arrays. For example, paste the following code into the Gosu Scratchpad.

```
var dateObject = new Date()
var propertyName = "Hour"

// Get a property name using reflection
print(dateObject[propertyName])
```

If the time is currently 5:00 AM, this code prints the following output.

```
5
```

The property name is case sensitive.

Retrieving methods by using reflection

You can use reflection to find all the methods that a class provides and to 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. Because of performance implications and the inability to find errors at compile time, only use reflection if there is no other way to do what you need.

To examine how reflection provides information about class methods, paste the following code into the Gosu Scratchpad.

```
var object = "This is a string"
var methodNames = ""
var methods = (typeof object).TypeInfo.Methods

for (method in methods) {
  methodNames = methodNames + method.Name + "  "
}

print(methodNames)
```

This code prints something lines to the following output, which is truncated for brevity.

```
wait()  wait( long, int )  wait( long )  hashCode()  getClass()  equals( java.lang.Object )
toString()  notify()  notifyAll()  @itype()  compareTo( java.lang.String )  charAt( int )
length()  subSequence( int, int )  indexOf( java.lang.String, int )  indexOf( java.lang.String )
indexOf( int )  indexOf( int, int )  codePointAt( int )  codePointBefore( int )
```

The following example gets a method by name and then calls that method. This example uses the `String` class and its `compareTo` method, which returns 0, 1, or -1. Paste the following code into the Gosu Scratchpad.

```
// Initialize test data
var objectText = "a"
var argumentText = "b"

// Get the list of methods
var methodInfos = String.Type.TypeInfo.Methods

// Find a specific method by name.
// More than one method signature may exist for the same name, so the return type is an array
// of results with each method having different argument numbers or types
var m = methodInfos.firstWhere( \ elt -> elt.DisplayName == "compareTo")
if (m == null) { throw "Could not find method by name" }

print("Full method 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].OwnersType.toString())

var res = m.CallHandler.handleCall( objectText, { argumentText } )
print("Call compareTo() using reflection: " + res)

var res2 = objectText.compareTo(argumentText)
print("Call compareTo() the normal way (with static typing): " + res2)
```

The code prints the following output.

```
Full method name is:   compareTo( java.lang.String )
Number of parameters is:   1
Name of first parameter is:   String
Call compareTo() using reflection: -1
Call compareTo() the normal way (with static typing): -1
```

Comparing types by using reflection

You can compare the types 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. This operator returns `false` if one type is a subtype of the other or if the types are in different packages.

Use the *`sourceType`*`.Type.isAssignableFrom(`*`destinationType`*`)` method to determine whether types are compatible for assignment. This method considers the possibility of subtypes, such as subclasses, which the equality operator does not. The method determines if the destination type argument is either the same as, or a superclass or superinterface of the source type.

The *`sourceType`*`.Type.isAssignableFrom(`*`destinationType`*`)` method examines only 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, although Gosu can convert `boolean` to `String` or from `String` to `boolean` using coercion, using those types with the `isAssignableFrom` method returns `false`.

Gosu provides a variant of this functionality with the Gosu `typeis` operator. The `typeis` operates between an object and a type. The *`type`*`.Type.isAssignableFrom(...)` operates between a type and another type.

For example, paste the following code into the Gosu Scratchpad.

```
var str: String = "hello" // Explicit declaration of "String" is optional, but is shown for clarity
var obj1: Object = "hello"
var obj2: Object = 2.345
var bool: Boolean = true

print("Typeof str: " + (typeof str).Name)
print("Typeof obj1: " + (typeof obj1).Name)
print("Typeof obj2: " + (typeof obj2).Name)

// The following line would be a compile error! String is not a subtype or supertype of Boolean.
// tempTypeIs = (bool typeis String)

print("String typeis Object: " + (str typeis Object) )

// Notice the different run time behavior of these two lines,
// Note that both obj1 and obj3 are Object at compile time,
// but have different run-time types
print("Object obj1 typeis String: " + (obj1 typeis String) )
print("Object obj2 typeis String: " + (obj2 typeis String) )
print("Boolean assignable from String: " + (typeof str).isAssignableFrom((typeof bool)))
print("String assignable from Boolean: " + (typeof bool).isAssignableFrom((typeof str)))
print("Object assignable from String: " + ((Object).Type.isAssignableFrom( String )) )
print("String assignable from Object: " + ((String).Type.isAssignableFrom( Object )) )

// For typical uses cases, DO NOT compare types with the == operator
// Using == to compare types returns false if one is a subtype.
// Instead, typically it is best to use the 'typeis' operator
print("Compare a string to object using ==: " + ( (typeof str) == Object ) )
```

This code prints the following output.

```
Typeof str: java.lang.String
Typeof obj1: java.lang.String
Typeof obj2: java.lang.Double
String typeis Object: true
Object obj1 typeis String: true
Object obj2 typeis String: false
Boolean assignable from String: false
String assignable from Boolean: false
```

```
Object assignable from String: true
String assignable from Object: false
Compare a string to object using ==: false
```

Java type reflection

Gosu implements a dynamic type system that is designed to be extended beyond its native objects. A dynamic type system enables Gosu to work with a variety of different types. This capability is not the same as items being dynamically typed. Gosu is a statically typed language.

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 differentiate between a Java class instance and a native Gosu object. Both sets of objects are exposed to the language through the same abstract type system. You can use Java types directly in your Gosu code. You can even extend Java classes by writing Gosu types that are subtypes of Java types. Similarly, you can implement or extend Java interfaces from Gosu.

The Gosu language transparently exposes and uses Java classes as Gosu objects through the use of Java `BeanInfo` objects. Java `BeanInfo` objects are analogous to Gosu's `TypeInfo` information. Both `BeanInfo` and `TypeInfo` objects encapsulate type metadata, including properties and methods. All Java classes have `BeanInfo` information either explicitly provided with the Java class or dynamically constructed at run time. Gosu examines a Java class's `BeanInfo` and determines how to expose this type to Gosu. Because of this behavior, your Gosu code can use the Gosu reflection APIs with Java types.

In rare cases, you might want to do reflection on the underlying implementation Java class of a Gosu type rather than the Gosu type itself. The Java implementation of the Gosu class is also known as the backing class. 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` as shown in the following example.

```
// Get the Java backing class
var javaClass = (theType as IHasJavaClass).BackingClass

// With that class, you can interate 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.

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 the fully qualified name of the type.

For example, the following code gets a type by a `String` version of its fully qualified name and instantiates it using the type information for the type.

```
var myFullClassName = "com.mycompany.MyType"
var type = TypeSystem.getByFullName( myFullClassName )
var instance = type.TypeInfo.getConstructor( null ).Constructor.newInstance( null )
```

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 other programming languages. Gosu feature literals are validated at compile time, preventing many common errors with reflection techniques. To refer to the feature of a type or object, you use the `#` operator after the type or object, followed by the feature name. The syntax is similar to the feature syntax of the Javadoc `@link` syntax.

You can use feature literals in APIs, such as the following types of layers.

- 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 references, you can access the underlying property or method information. Alternatively, use feature literal references to invoke a method or access 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.

The following example code uses feature literal syntax to get and set property values on the `Employee` class defined in the previous section.

```
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 ) // Prints "Ed"
```

Calling a method by using feature literal syntax

The following example code uses feature literal syntax to call a method on the `Employee` class defined earlier in this topic.

```
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)
```

Binding a property to an instance by using feature literal syntax

You can bind a feature literal to a specific object instance. This technique is the equivalent of using the compile-time type to request the property by name for a particular instance of the object.

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.

Binding argument values by using feature literal syntax

You can bind argument values in method references that use feature literal syntax.

```
var anEmp = new Employee() { :Name = "Joe", :Age = 32 }
var updateFuncForAnEmp = anEmp#update( "Ed", 34 )

print( anEmp.Name ) // Prints "Joe". Code has not yet invoked 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 by using feature literal syntax

Feature literals support type-safe chaining, so this code is valid.

```
var bossesNameRef = anEmp#Boss#Name
```

The chained feature literal refers to the name of the boss of the object in the `anEmp` variable.

Using feature literal syntax to convert method references to blocks

You can convert method references to blocks with the `toBlock` method.

```
var aBlock = anEmp#update( "Ed", 34 ).toBlock()
```

chapter 26

Concurrency

This topic describes Gosu APIs that protect shared data from access from multiple threads.

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 called *concurrency*. Code that is designed to safely get or set concurrently accessed data is called *thread safe*.

The most common situation that requires concurrency handling is the use of data in class static variables. Static variables are variables that are stored once for a class rather than once for each instance of the class. If multiple threads on the same Java virtual machine access this class, you must ensure that simultaneous accesses to this data safely get or set the data.

If you are certain that static variables or other shared data are 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. For example, if you need to manage a single local memory cache that multiple threads use, you must carefully synchronize all reads and writes to shared data.

WARNING Static variables can be extremely dangerous in a multi-threaded environment. Using static variables in a multi-threaded environment can cause problems in a production deployment if you do not properly synchronize access. If such problems occur, they are extremely difficult to diagnose and debug. Timing in an multi-user multi-threaded environment is difficult, if not impossible, to control in a testing environment.

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. Your plugin code must have the following attributes:

- 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, PolicyCenter 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. Do not assume shared data or temporary storage is unique to one user request, which is one HTTP request from a single user.

Gosu provides the following types of concurrency APIs to support writing 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.

Lazy concurrent variables

The `LockingLazyVar` class (in `gw.util.concurrent`) implements what is known as a lazy variable. Gosu does not construct a *lazy variable* until the first time any code uses the variable. 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 type-safe holder.

Concurrent cache

The `Cache` class in `gw.util.concurrent` declares a cache of values that 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.

WARNING Caches can cause subtle problems. Use caches only as a last resort for performance.

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 `using` clauses provide access to these classes and properly handle cleanup if exceptions occur. Gosu also provides a readable syntax for creating custom objects for reentrant object handling.

Concurrency API does 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. PolicyCenter includes built-in systems to synchronize and intelligently cache entity data across a cluster.

Do not attempt to synchronize Guidewire entities

Never try to synchronize access to Guidewire entities defined in the data model configuration files. PolicyCenter automatically manages synchronized access and entity loading and caching, both locally and across a cluster if you use clusters.

A transaction bundle based on the class `gw.transaction.Bundle` is not thread-safe. Be aware of any concurrency issues when accessing entity instances in a bundle or committing those instances to the database. If multiple users can access the entity instances in a bundle simultaneously, you must use external synchronization to ensure reading and writing correct data, such as for property values.

WARNING Use external synchronization to ensure data integrity for concurrent access to entity instances in a transaction bundle.

See also

- “Request and session scoped variables” on page 419
- “Concurrent lazy variables” on page 420
- “Concurrent cache” on page 421
- “Concurrency with monitor locks and reentrant objects” on page 422
- “Bundles and database transactions” on page 387
- *Integration Guide*

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 life cycle 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.

In the base configuration, these classes do not work in the context of web service implementation or Gosu servlet classes. To enable web services and Gosu servlets as valid sources for variables in a `RequestVar` object, enable the `ScopedVariableSupportedForAllCapturedRequests` configuration parameter.

For both objects, Gosu provides a `RequestAvailable` property. Check the value of this `boolean` property to see if the `RequestVar` object has a request or the `SessionVar` object has a session 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 the property. 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 having a value of `null`.

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, many 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 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 sets this property:

```
MyClass.MySessionVar = "hello"
```

In another part of the application, you can write code that gets this property:

```
print( MyClass.MySessionVar )
```

This example explicitly checks the `RequestAvailable` property. In business code, for example to support your batch processes, you must decide what to do if `RequestAvailable` returns `false`.

Using `RequestVar` and `SessionVar` is strongly recommended, rather than the Java thread local API `java.lang.ThreadLocal<`*TYPE*`>`. Because application servers pool their threads, using `ThreadLocal` increases the

risk of data remaining forever as a memory leak, because the thread to which you attached it never terminates. Also, pooled threads can preserve stale data from a previous request, because 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 with a `try/finally` block. Better practise is to convert that code to use `RequestVar` and `SessionVar`.

See also

- "Generics" on page 221
- *Integration Guide*

Concurrent lazy variables

In addition to using the Java native concurrency classes, Gosu includes utility classes that provide additional concurrency functionality. The `LockingLazyVar` class implements what is known as a lazy variable. Gosu constructs a *lazy variable* the first time some code uses the variable and not at the variable declaration. 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 type-safe holder.

In Gosu, the `LockingLazyVar.make(gw.util.concurrent.LockingLazyVar.LazyVarInit)` method signature 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. The following example demonstrates the use of this method and clarifies the method signature:

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

The example passes a block as an argument to `LockingLazyVar.make(...)`. That block creates a new `ArrayList` that is parameterized to the `String` class. The parameter is a block that creates a new object. In this case, the block returns a new `ArrayList`. You can create any object. In your business-use code, this block might be very resource-intensive in creating or loading this object.

It is best to let Gosu infer the correct type of the block and the result of the `make` method, as shown in this example. Using Gosu type inference simplifies your code because you do not need to use explicit Gosu generics syntax to define the block type, such as the following version:

```
var _lazy : LockingLazyVar<List<String>> = LockingLazyVar.make( \-> new ArrayList<String>() )
```

To use the lazy variable, call its `get` method:

```
var i = _lazy.get()
```

If the block has not yet run, Gosu runs the block when you access the lazy variable. If the block has already run, Gosu returns the cached value of the lazy variable and does not rerun the block.

A good approach to using a lazy variable is to define the variable as static and then define a property accessor function to abstract the implementation of the variable. The following code shows 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. Gosu runs the very expensive method only once, the first time that code

accesses the lazy variable value. If any code accesses this property again, Gosu uses the cached value and does not execute the block again. This behavior is useful in cases where you want a system to start quickly and pay only 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 this class does not lock, it performs faster and has less chance of server deadlock. However, this class is unsafe in any context that has potential concurrent access and thus requires thread safety.

Concurrent cache

A similar class to the `LockingLazyVar` class is the `Cache` class. An instance of this class declares a concurrent cache of values that you can look up quickly and in a thread-safe way. The cache is similar to a Least Recently Used (LRU) cache. Because the `Cache` class uses the Java concurrency libraries, access to the cache is thread-safe. Each entry in the cache is defined by a key, which is also called an input value. A block uses the value of the key to calculate the value of the cache entry.

The constructor for a cache requires:

- A name as a `String`. The implementation uses this name to generate logging for cache misses.
- The size, as a number of slots.
- A block that defines a function to calculate a value from a key. Typically, this calculation is resource-intensive.

Use the key and value types to parameterize the `Cache` type using Gosu generics syntax. For example, if you need to pass a `String` to the cache and get an `Integer` back, create a new `Cache<String, Integer>`.

To use the cache, call the `get` method and pass an input value, which is the key. If the value for the key is in the cache, `get` returns that value. If the value is not cached, Gosu calls the block to calculate the value from the key and then caches the result. On any subsequent call of the `get` method, the value is the same but Gosu uses the cached value, if it is still in the cache. If too many items were added to the cache and the required item is unavailable, Gosu reruns the block to regenerate the value. Gosu then caches the result again.

To use a cache within another class, you can define a static instance of the cache. The static variable definition defines your block. Because the `Cache` class uses the Java concurrency libraries, this static implementation is thread-safe. For example, in your class definition, define a static variable like this:

```
static var _myCache = new Cache<String, Integer>( "StrToInt", 1000, \ str -> getMyInt( str ) )
```

To use your cache, 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 = _myCache.get( inputString )
```

The first time you call the `get` method, Gosu calls the block to generate the `Integer` value.

On any subsequent call of the `get` method, the value is the same but Gosu uses the cached value, if it is still in the cache. If too many items were added to the cache and the required item is unavailable, Gosu reruns the block to regenerate the value. Gosu then caches the result again in the concurrent cache object.

An even better way to use the cache is to abstract the cache implementation into a property accessor function. A private static object `Cache` object, as shown in the previous example, can handle the actual cache. For example, define a property accessor function such as:

```
static property get function MyInt( str : String ) {
  return _myCache.get( str )
}
```

These code examples are demonstrations that have a simple operation in the block. Typically, the overhead of maintaining the cache is appropriate if your calculation is resource-intensive and you expect repeated access with the same input values.

WARNING Incorrect implementation of caching can lead to run-time errors and data corruption. Only use caches as a last resort for performance issues.

Create a thread-safe cache

Procedure

1. Decide the key and value types for your cache based on input data.

   For example, you need to pass a `String` and get an `Integer` back from the cache.
2. Construct a new cache.

   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() )
```

3. To use the cache, call the `get` method and pass the input value, which is the key. If the value is in the cache, `get` returns the value from the cache. If the value is not cached, Gosu calls the block and calculates the value from the key and then caches the result.

   For example:

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

   This code prints:

```
"HELLO WORLD"
"HELLO WORLD"
```

Result

In this example, the first call of the `get` method calls the block to generate the uppercase value. On the second call of the `get` method, the value is the same but Gosu uses the cached value.

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 a variable's data to prevent simultaneous access to the data.

The most straightforward implementation of a lock is to define a static variable for the lock in your class definition. Next, define a property get accessor function that uses the lock and calls another method to perform the task that you must synchronize. This approach uses a Gosu `using` clause with reentrant objects to simplify concurrent access to shared data. The `using` statement syntax for lock objects causes Gosu to consider the object reentrant.

A *reentrant* object supports multiple concurrent uses of a single lock in a single thread. Reentrancy maintains a count of the number of times the thread sets and releases the lock. In reentrant lock use, the thread does not deadlock on setting an already locked lock, nor release the lock when other code in the thread still retains that lock. Reentrant objects help manage safe access to data that is shared by reentrant 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 at least one of the following attributes:

- The object implements the `java.util.concurrent.locks.Lock` interface. For example, the following Java classes in that package implement this interface: `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.
- The object implements the Gosu class `gw.lang.IReentrant`. This interface contains two methods with no arguments: `enter` and `exit`. Your implementation 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 that use locks by implementing `java.util.concurrent.locks.Lock`, a `using` clause simplifies the code. The `using` statement always cleans up the lock, even if the code throws an exception.

```
uses java.util.concurrent

// 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 code uses the object itself, by using the keyword `this`, as the monitor lock:

```
function monitorLock() {
  using ( this as IMonitorLock ) {
    // Do stuff
  }
}
```

This approach is equivalent to a `synchronized` block in the Java language.

The following code uses the `java.util.concurrent.locks.ReentrantLock` class using more verbose `try` and `finally` statements. This approach is not recommended:

```
// In your class variable definitions...
var _lock : ReentrantLock = new ReentrantLock()

function useReentrantLockOld() {
  _lock.lock()
  try {
    // Do your main work here
  }
  finally {
    lock.unlock()
  }
}
```

Alternatively, you can do your change in a Gosu block. This approach is not recommended. Returning the value from a block imposes more restrictions on how you implement `return` statements. It is typically better to use the `using` statement structure shown earlier in this topic.

```
uses java.util.concurrent

...

property get SomeProp() : Object {
  var retValue : Object
  _lock.with( \-> {
    retValue = _someVar.someMethod()
  })
  return retValue
}
```

See also

- “Object life-cycle management with using clauses” on page 122
- http://en.wikipedia.org/wiki/Monitor_(synchronization)
- http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

chapter 27

Checksums

This topic describes APIs for generating checksums. A *checksum* is a calculated number to detect modification of data in transit. Such modification can be either accidental or malicious. For example, you can use a checksum to detect corrupted stored data or errors in a communication channel. Longer checksums such as 64-bit checksums are also known as *fingerprints*.

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 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 in a way that preserves its checksum value or, depending on transport integrity, substitute a new checksum. To guard against malicious changes, use encryption at the data level with a cryptographic hash or the transport level, such as SSL/HTTPS.

WARNING Checksums improve detection from accidental modification of data but cannot detect all 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 synchronizing 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 resynchronize the data. To work effectively, the fingerprint algorithm must provide near certainty that a real-world change would change the fingerprint. Having two different values generate a matching fingerprint is called a *collision*. A fingerprint uniquely identifies the data for most practical purposes, although changed data causing a collision 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

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 where `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. The following mathematical notation demonstrates this assumption, given two `String` variables `s1` and `s2`, using the → symbol to mean "implies":

```
new FP64(s1).equals(new FP64(s2)) → s1.equals(s2)
```

Do not fingerprint the value of a fingerprint by fingerprinting the output of the `FP64` methods `toBytes` and `toHexString`. Using this type of fingerprint might lead to unexpected collisions. The shorter length of the fingerprint itself invalidates the probabilistic guarantee that a collision is unlikely to occur.

Creating a fingerprint

To create a fingerprint object, instantiate the `gw.util.fingerprint.FP64` object and pass one of the supported object types to the constructor:

- A `String` object:

  ```
  var s = "hello"
  var f = new FP64(s)
  ```

- A character array:

  ```
  var s = "hello"
  var ca : char[] = {s[0], s[1], s[2], s[3], s[4]}
  var f = new FP64(ca)
  ```

  An alternative method signature takes extra parameters for start position and length of the desired series of characters in the array.

- A byte array:

  ```
  var ba = "hello".Bytes // Or use "hello".getBytes(
  var f = new FP64(ba)
  ```

  An alternative method signature takes extra parameters for start position and length of the desired series of byes in the array.

- A stream object:

  ```
  var s = "testInputStreamConstructor"
  new FP64(new ByteArrayInputStream(gw.util.StreamUtil.toBytes(s))));
  ```

- An input stream:

  ```
  var s = "testInputStreamConstructor"
  new FP64(new StringBuffer(g));
  ```

- Another `FP64` fingerprint object to duplicate the fingerprint:

  ```
  var s  = "hello"
  var f  = new FP64(s)
  var f2 = new FP64(f)
  ```

Generating the fingerprint binary data

To generate output data from a fingerprint, 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 alternative method signature that takes a byte array buffer into which the method writes the bytes. The buffer must have a length of 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.

Fingerprints as keys in hash maps

Because the `FP64` class overrides the Java methods `equals` and `hashCode`, you can use `FP64` objects as keys in hash tables. For example, use `FP64` values as keys in a `java.util.HashMap` instance.

Extending fingerprints

The `FP64` class provides methods for extending an existing fingerprint by more bytes or characters. This functionality is useful if the only change to the source data was appending a known series of bytes to the end of the original `String` data.

To produce a fingerprint equivalent to the fingerprint of the concatenation of two `String` objects, you can extend the fingerprint created from one `String` using another `String`. Given the two `String` variables `s1` and `s2`, the following is true:

```
new FP64(s1 + s2).equals( new FP64(s1).extend(s2) )
```

The same logic is true for character arrays, not just `String` objects.

All operations for extending a fingerprint are destructive. The operations modify the fingerprint object directly, in place. All operations return the resulting `FP64` object, so you can chain method calls together, such as in the following code:

```
new FP64("x").extend(foo).extend(92))
```

If you need to make a copy of a fingerprint, instantiate the `FP64` object and use the `FP64` object to copy as the constructor argument:

```
var original = new FP64("Hello world")
var copy = new FP64(original) // A duplicate of the original fingerprint
```

part 3

Gosu programs

chapter 28

Command-prompt tool

A *Gosu program* is a file with a `.gsp` file extension that you can run directly from a command-prompt tool. You can run self-contained Gosu programs outside the PolicyCenter server by using the Gosu command-prompt tool. The Gosu command-prompt tool encapsulates the Gosu language engine.

Gosu command-prompt tool basics

You can use the Gosu language command-prompt tool outside the PolicyCenter server to perform the following tasks:

- Invoke Gosu programs (`.gsp` files), which can use Gosu classes, Gosu extensions, and Java classes
- Evaluate Gosu expressions passed to the command prompt.

Accessing entities and other types from the Gosu command-prompt tool

Gosu programs that you run with the Gosu command-prompt tool cannot access some types available from PolicyCenter Studio. For example, you cannot access entity types, PCF types, and plugin types. Some types have different sets of methods because some Gosu enhancements are unavailable in the command-prompt tool.

IMPORTANT Guidewire does not support the import of built-in PolicyCenter application JARs into Gosu command-prompt tool programs.

To add, edit, delete, or query PolicyCenter entity instances from a Gosu program, implement the majority of your code as a PolicyCenter WS-I web service. From your Gosu program, read the command-prompt arguments, and then call the web service.

See also

- "Command-prompt arguments" on page 433
- *Integration Guide*

Unpacking and installing the Gosu command-prompt tool

The PolicyCenter main product installation includes the Gosu tool as a subdirectory named `admin`. The `admin` directory includes the following files and directories:

Path	Purpose
`/bin/gosu.cmd`	The Windows tool that invokes Gosu

Path	Purpose
`/src/gw/.../*.gs`	Core Gosu classes
`/src/gw/.../*.gsx`	Core Gosu enhancements
`/lib/*.jar`	Java archive (JAR) files that contain core Gosu libraries

To use the Gosu tool on another computer, copy the entire `admin` directory to that computer. Ensure that the computer has a supported version of the Java run time.

IMPORTANT You are licensed to use the Gosu tool only to work with Guidewire applications.

You can change your system's `path` to add the Gosu tool `bin` directory so that the command prompt can find the `gosu` command. On Windows, modify the system `Path` variable by going to the **Start** menu and choosing **Control Panel→System→Advanced System Settings→Environment Variables**. In **System variables**, choose the `Path` variable. At the end of the variable value, type a semicolon and the full path to the `bin` directory in the Gosu tool directory.

For example, suppose you installed the Gosu shell directory to the path:

```
C:\gosu\
```

Add the following to the system path:

```
;C:\gosu\bin
```

To test this path, close any existing command-prompt windows, and then open a new command-prompt window. Type the following command:

```
gosu -help
```

If the Gosu help information appears, the Gosu tool is installed correctly.

Command-prompt tool and options

To run the Gosu command-prompt tool, you use the `gosu` command.

The following table lists the command-prompt tool parameters and options.

Task	Options	Example
Default behavior of the command prompt tool with no options is to run Gosu Lab.		`gosu`
Run a Gosu program. Include the `.gsp` file extension when specifying the file name.	*`filename`*`.gsp`	`gosu myprogram.gsp`
Enable checking for numeric overflow in arithmetic operations. This option requires a Gosu program to run.	`-checkedArithmetic` *`filename`*`.gsp`	`gosu -checkedArithmetic myprogram.gsp`
Add more paths to the search path for Java classes or Gosu classes. Separate paths with semicolons.	`-classpath` *`path`*	`gosu -classpath C:\gosu \projects\libs`
Evaluate a Gosu expression at the command prompt. Surround the entire expression with quotation marks. For any quotation mark in the expression, replace it with three quotation marks. For other special DOS characters such as > and <, precede them with a caret (^) symbol.	`-e` *`expression`* `-eval` *`expression`*	`gosu -e "new java.util.Date()"` `gosu -e """a"""+"""b"""`
Print help information for this tool.	`-h` `-help`	`gosu -h`

Task	Options	Example
Show Gosu version	-v -version	gosu -version

See also

- "Checked arithmetic for add, subtract, and multiply" on page 80

Write and run a Gosu program

You can run a Gosu program from a command prompt. The Gosu program uses the same keywords and syntax as a Gosu class.

Before you begin

The following instructions describe running a basic Gosu program after you install the `gosu` command-prompt tool. These instructions apply only to the command-prompt tool for Gosu. If you are using the Gosu plugin for the IntelliJ IDEA IDE, to get command-prompt Gosu, you must download the full distribution from:

`http://gosu-lang.org/downloads.html`

Procedure

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 that contains your program.
4. Type the following command:

```
gosu myprogram.gsp
```

   If you have not yet added the `gosu` executable folder to your system path, instead type the full path to the `gosu` executable.

   The tool runs the program.

   The program prints the following output:

```
Hello World
```

See also

- "Unpacking and installing the Gosu command-prompt tool" on page 431

Command-prompt arguments

You can access command-prompt arguments to Gosu programs by manipulating raw arguments. You can parse the full list of arguments from the command line as positional parameters. Alternatively, you can write a custom parser to identify named options and their values. For example, you can write a parser for an option that has multiple parts that are separated by space characters, such as `-username jsmith`. In this example, each of the `-username` and the `jsmith` components is a separate raw argument.

To get the full list of command-prompt arguments as a list of `String` values, use the `RawArgs` property of the `gw.lang.Gosu` class. This property provides an array of `String` values. You can access the values in this array in the same way as any other array. For example, you can use Gosu code like the following lines.

```
// TestArgs.gsp
var myArgs = Gosu.RawArgs
```

```
var numArgs = myArgs.Count
print("${myArgs} ${numArgs}")
for (str in myArgs) {
  print("${str}")
}
```

You can run the program with the following command.

```
TestArgs one two three
```

This code prints the following lines.

```
[one, two, three] 3
one
two
three
```

Create and run a Gosu program that processes arguments

A Gosu program that you run from the command prompt can read arguments from the command line and report errors in the values of those arguments.

Procedure

1. Choose a directory in which to save your command-prompt tool. In this directory, create a subdirectory called `test`, which is the package name.
2. Create a Gosu class that defines your properties. The following code defines two properties, one `String` property named `Name` and a `boolean` property named `Hidden`:

```
package test

class Args {
  // String argument
  static var _name : String as Name = "No name"

  // boolean argument -- No value to set on the command line
  static var _hidden : boolean as Hidden = false
}
```

3. Save this Gosu class file as the file `Args.gs` in the `test` directory.
4. Create a Gosu program, using the following code:

```
uses test.*
var args = Gosu.RawArgs
var numArgs = args.Count

try {
  if (arg.startsWith( "-" )) {
    if ((arg == "-name") and (i< numArgs-1)) {Args.Name = args[i+1]}
    if (arg == "-hidden") {Args.Hidden = (i< numArgs-1) and (args[i+1] == "false")?false:true}
  }
} catch (Exception) {
  print("Error parsing args: " + args)
}

print("hello " + Args.Name)
print("you are " + (Args.Hidden ? "hidden" : "visible") + "!!!!")
```

5. Click **Save As** and save this new command-prompt tool as `myaction.gsp` in the directory that contains the `test` subdirectory.
6. Open a command-prompt window and navigate to the directory that contains `myaction.gsp`.
7. In the command-prompt window, enter the following command

```
gosu -classpath "." myaction.gsp -name John -hidden
```

This command prints:

```
hello John
you are hidden!!!!
```

chapter 29

Programs

A *Gosu program* is a file with a `.gsp` file extension that you can run directly from a command-prompt tool.

You can run self-contained Gosu programs outside the PolicyCenter server by using the Gosu command-prompt tool. The Gosu shell command-prompt tool encapsulates the Gosu language engine. You can run Gosu programs directly from the Windows command prompt as an interactive session or run Gosu program files.

The following instructions describe running a basic Gosu program after you install the Gosu command-prompt tool. These instructions apply only to the command-prompt shell for Gosu. If you are using the Gosu plugin for the IntelliJ IDEA IDE, to get command-prompt Gosu you must download the full distribution at:

```
http://gosu-lang.org/downloads.html
```

See also

- "Command-prompt tool" on page 431

The structure of a Gosu program

A Gosu program includes one or more lines that contain Gosu statements. A Gosu program can also include the following elements:

- A metaline
- One or more arguments from the command prompt
- One or more functions

See also

- "Command-prompt arguments" on page 433
- "Create and run a Gosu program that processes arguments" on page 434

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 line is for compliance with the UNIX standard for shell script files. The metaline is optional. If present, this line must be the first line of the program. The metaline looks like the following:

```
#!/usr/bin/env gosu
```

The # character that begins the metaline does not start a comment line in Gosu programs. The # character is not a valid line-comment start symbol.

Functions in a Gosu program

Your Gosu program can define functions in the same file and call those functions.

For example, the following program creates a function and calls the function twice:

```
print(sum(10, 4, 7));
print(sum(222, 4, 3));

 function sum(a: int, b: int, c: int) : int {
  return a + b + c;
}
```

Running this program prints:

```
21
229
```

part 4

Testing

chapter 30

GUnit Test Framework

The GUnit Test Framework enables the writing of tests to exercise Gosu configuration code.

Tests are written in Gosu. You can execute these tests from the command line to include the tests in an automated continuous integration work flow. You can also execute these tests from Guidewire Studio.

Create a framework test

The GUnit Test Framework supports several types of tests, each of which is based on one of the following classes.

`PCServerTestClassBase`
Base class for tests that require a running server

`PCUnitTestClassBase`
Base class for tests that do not need the services provided by a running server

A test class extends one of the base classes. The test class name must end with the suffix `Test`, as in `MySampleTest`. The base class implementation of the framework's `setUp` method tests for this condition and throws an `IllegalStateException` if it is not met.

Store all test class files within the Studio `modules/configuration/gtest` directory hierarchy. New subdirectories can be created within the hierarchy to organize test classes for your convenience.

```
@Suites("UniqueSuiteName")
class MyTestClass extends PCUnitTestClassBase {
  ...
}
```

Optionally, the `@Suites` annotation can be specified on the test class. The `@Suites` annotation accepts a unique `String` argument. All the test classes grouped in a particular suite must specify the same `@Suites` annotation. An associated suite class will subsequently be defined that also references the suite's string value.

The following sample source code demonstrates the use of the `@Suites` annotation. The `MySuiteNames.gs` file defines a string constant called `SAMPLE_SUITE_NAME`. In the `MyTestClass.gs` file, the `MyTestClass` class is defined with a `@Suites` annotation that specifies the `SAMPLE_SUITE_NAME` constant. If other test classes existed in the suite, each class would specify the same `@Suites` annotation. Finally, the `MySuiteClass.gs` file defines a suite class that references `SAMPLE_SUITE_NAME`. Details about defining a suite class are described in a subsequent topic.

```
// ========== File: MySuiteNames.gs ==========
// Define suite name strings.
// Test and suite classes reference the same string to group the tests into the suite.
// A test class references the string in a @Suites annotation.
```

```
package gw.suites

@Export
class MySuiteNames {

  public static final var SAMPLE_SUITE_NAME : String = "AnyUniqueString_1"
  // ... Define other suite-name strings here
}

// ========== File: MyTestClass.gs ==========
// Implement test class

package gw.suites

uses gw.suites.MySuiteNames

@Suites(MySuiteNames.SAMPLE_SUITE_NAME)
class MyTestClass extends PCUnitTestClassBase {
  ...
}

// ========== File: MySuiteClass.gs ==========
// Implement suite class

package gw.suites

uses gw.suites.MySuiteNames

class MySuiteClass {
  // References MySuiteNames.SAMPLE_SUITE_NAME.
  // All test classes with a @Suites annotation that references SAMPLE_SUITE_NAME are included in the suite.
  // Details of implementing the suite class are described in a subsequent topic.
}
```

The `@Suites` annotation is optional. Similarly, a suite class is not required to reference a unique suite name. A suite class that does not reference a suite name will include all the test classes in the package that do not specify a `@Suites` annotation. This "catch all" style of suite grouping can be useful early in the development cycle when only a small number of tests exists and a logical criteria for grouping them into suites is not yet evident.

Each test class provides the following Gosu constructors.

```
construct()
construct(name : String)
```

The name of a framework test method must begin with the string `test` as in `testAddNumbers`. When running the individual tests in a test class, the framework automatically executes each method that begins with the string `test`.

The following code illustrates the minimal skeletal structure of a test class based on the `PCServerTestClassBase` class.

```
class MyTestClass extends PCServerTestClassBase {

  construct(testName : String) {
    super(testName)
    ...
  }

  function testBasicSystemCheck() {
    ...
  }

  function testAnotherServerTest() {
    ...
  }

  // ... Other class functions

}
```

Core functions

The GUnit Test Framework base classes provide core functions like `setUp`, `beforeMethod`, `afterMethod`, and `tearDown` that define the basic skeletal structure of a test. The core functions are called automatically by the framework as it processes the individual tests in a test class and test suite.

This section presents the core functions in the order in which they are executed when running a series of tests in a test class and test suite. The following pseudocode further illustrates the execution order of each core function.

```
for each test class in a test suite {
  construct()
  beforeClass()

  for each test in a test class {
    setUp()
    beforeMethod()

    // Actual test method, such as testCheckSystem()

    afterMethod()
    tearDown()
  }

  afterClass()
}
```

Constructors

```
construct()
construct(name : String)
```

The `name` property is optional and not used by the framework. It can be used by a test or test suite to identify a particular test object. If the name is not initialized in the constructor, it can be assigned by calling the object's `setName` method.

Method: beforeClass

```
beforeClass() : void
```

The `beforeClass` method is executed a single time before running any of the test methods defined in a particular test class. In a test suite with multiple test classes, each test class's `beforeClass` method is executed immediately before running the tests in the class.

Do not create test instance variables in the `beforeClass` method; the variables will exist only for the first executed test. Instead, create instance variables in `beforeMethod`.

```
override function beforeClass() {
  super.beforeClass()
  ...
}
```

Method: setUp

```
final setUp() : void
```

The `setUp` method is executed before running each test method defined in a test class. The method typically configures the test context and creates any data objects required by the test.

A test class can overload the `setUp` method, but cannot override it.

In the base class implementation, if the test object's class name does not end with the string `Test` then an `IllegalStateException` is thrown.

```
function setUp(someArg : int) {
  super.setUp()
```

```
  ...
}
```

Method: beforeMethod

```
beforeMethod() : void
```

The `beforeMethod` method is executed before running each test method defined in the test class.

```
override function beforeMethod() {
  super.beforeMethod()
  ...
}
```

Method: afterMethod

```
afterMethod(possibleException : Throwable) : void
```

The `afterMethod` method is called after each test method completes its execution.
The base class implementation clears all instance variables of the completed test.

```
override function afterMethod(possibleException : Throwable) {
  super.afterMethod(possibleException)
  ...
}
```

Method: tearDown

```
final tearDown() : void
```

The `tearDown` method is called after each test method completes its execution. The method typically performs clean-up operations.
A test class can overload the `tearDown` method, but cannot override it.

```
function tearDown(someArg : int) {
  super.tearDown()
  ...
}
```

Method: afterClass

```
afterClass() : void
```

The `afterClass` method is called after the completion of all tests in a test class.

```
override function afterClass() {
  super.afterClass()
  ...
}
```

Support functions

The GUnit Test Framework provides several categories of support functions that tests can call to implement a desired test operation.

Testing environment

The framework provides functions to configure the testing environment and return information about it.

Method: assert...

The framework provides various methods that assert whether a particular condition exists. Each method's name begins with the prefix `assert` followed by the condition tested, as in `assertEquals` and `assertNotZero`.

The framework supports all the methods defined by the `Assert` class in the Java JUnit unit testing framework. The `Assert` class provides commonly used methods like `assertEquals`, `assertTrue`, and `assertFalse`. For a complete list of supported methods, refer to the JUnit `Assert` class documentation.

In addition, the framework extends the `Assert` class methods with new methods applicable to testing InsuranceSuite configuration code. Example methods include `assertBigDecimalEquals` and `assertDateEquals`. The framework assert methods are defined in the `PLAssertions` class, which is included in the `gw.testharness.v3` package.

The following table groups the methods into general categories. For complete details, refer to the `PLAssertions` class documentation.

Category	Methods
BigDecimal	assertBigDecimalEquals, assertBigDecimalNotEquals, assertBigDecimalIsZero, assertBigDecimalIsNotZero
Collection	assertEmpty, assertSize, assertCollectionContains, assertCollectionDoesNotContain, CollectionEquals, CollectionSame
Equals	assertEquals, assertEqualsIgnoreCase, assertEqualsIgnoreLineEnding, assertEqualsIgnoreWhiteSpace, assertEqualsReplaceAll, assertEqualsUnordered, assertComparesEqual
General purpose	assertDateEquals, assertFalseFor, assertTrueWithin, assertGreaterThan, assertZero, assertNotZero
Hashtable	assertHashtableContains, assertHashtableContainsKey
Helper	assertCollection, assertList, assertSet, assertThat
Iterator	assertIteratorEquals, assertIteratorSame
List	assertListEquals, assertListSame, findObjectInListUsingComparator
Miscellaneous	assertAssignable, assertExceptionThrown, assertExceptionThrownWithMessage, assertMethodDeclaredAndOverridesBaseClass
Object	assertArrayContains, assertArrayDoesNotContain, assertLength

Method: getTestResultsDir

```
getTestResultsDir() : File
```

The `getTestResultsDir` method returns a `java.io.File` object that references the directory where the test results are stored.

Method: registerPlugin

```
registerPlugin<T extends InternalPlugin>(pluginInterface : Class<T>, implementation : T) : void
```

The `registerPlugin` method registers a replacement plugin implementation to be used during the class's testing operations. The method is available only in test classes that extend `PCServerTestClassBase`.

The `pluginInterface` argument specifies the plugin to be replaced. The `implementation` argument references the plugin implementation to use during the testing operations.

The original plugin implementation is automatically restored at the completion of the class's tests by the base class implementation of the `afterClass` method.

```
class MySampleTest extends PCServerTestClassBase {

  // ... Constructors, other functions, etc.

  // *** Replacement plugin implementation for testing operations
```

```
  class MyReplacementPlugin implements IGenericPlugin {

    // ... Constructors, overrides of plugin functions, etc.
  }

  // *** Function to test my configuration code
  // The replaced plugin is subsequently restored automatically by the afterClass() method
  function testMyConfigCode() {

    registerPlugin(IGenericPlugin, new MyReplacementPlugin())    // Register replacement plugin for testing operations

    // ... Perform tests here
  }

  // *** Another test function
  // At the function's completion, restore the original plugin; don't wait until afterClass()
  function testMyConfigCode02() {
    var savedPlugin : IGenericPlugin

    savedPlugin = Plugins.get(IGenericPlugin.class)              // Save original plugin implementation
    registerPlugin(IGenericPlugin, new MyReplacementPlugin())    // Register replacement plugin

    // ... Perform tests here

    registerPlugin(IGenericPlugin, savedPlugin)                  // Restore original plugin
  }
}
```

Method: setUpMutableSystemClock

```
setUpMutableSystemClock() : void
```

The `setUpMutableSystemClock` method establishes a temporary system clock available for testing purposes. Test code can adjust the time of the temporary clock without affecting the actual system clock. The method is available only in test classes that extend `PCServerTestClassBase`.

Call the `setUpMutableSystemClock` method in the test class's `beforeClass` method. The temporary clock is initialized to the current system day and time. The original system clock is automatically restored by the base class implementation's `afterClass` method.

The `ChangesCurrentTimeUtil` class provides static methods to set and advance the temporary clock. In general, time is advanced to the future. Reversing time to the past can result in unexpected application behavior and is strongly discouraged.

```
setCurrentTime(test : TestCase, timeInMillis : long) : void
incCurrentTime(test : TestCase, deltaInMillis : long) : void
```

The `setCurrentTime` method sets the temporary system clock to a specified time. The `incCurrentTime` method increments the clock by a specified number of milliseconds.

The `test` argument references the test class. The `timeInMillis` argument specifies the time in milliseconds to assign to the clock. The `deltaInMillis` argument specifies the number of milliseconds to advance the clock.

```
// Set the temporary system clock to a future date
ChangesCurrentTimeUtil.setCurrentTime(this, mySampleCalendar.getSampleEventDateTime())

// Advance the clock by one day
var ONE_DAY_IN_MILLISECONDS = 1000L*60L*60L*24L
ChangesCurrentTimeUtil.incCurrentTime(this, ONE_DAY_IN_MILLISECONDS)
```

Logging

The framework provides functions that support logging. The logging functions are based on the `org.slf4j.Logger` object. Refer to the SLF4J (Simple Logging Facade for Java) documentation for details.

Method: addLoggingAppender

```
addLoggingAppender(logger : Logger, appender : LogAppender) : void
```

The `addLoggingAppender` method adds a logging output destination to a `Logger` object.
The `logger` argument references the `Logger` object to receive the new output destination. The `appender` argument specifies the new logging output destination. The `LogAppender` object is based on Log4j. Refer to the Log4J documentation for details on using an appender to define a new output destination.

Method: getLogger

```
getLogger() : Logger
```

The `getLogger` method returns the `org.slf4j.Logger` object used by the test framework for logging.

Method: setLoggerLevel

```
setLoggerLevel(logger : Logger, level : LogLevel) : void
```

The `setLoggerLevel` method sets the types of messages to log.
The `logger` argument specifies the `Logger` object used by the test framework. The object can be retrieved by the `getLogger` method.
The `level` argument specifies the log level. The following hierarchical levels are supported. Each level includes the levels below it. For example, the `INFO` level will log messages of the types `INFO`, `WARN`, and `ERROR`.

- `ALL` - Log all message types.
- `TRACE` - Verbose logging mode.
- `DEBUG` - Log debugging messages.
- `INFO` - Log informational messages about executing operations. INFO is the default log level.
- `WARN` - Log potential problems.
- `ERROR` - Log error conditions.
- `OFF` - Disable logging.

```
uses gw.logging.LogLevel

setLoggerLevel(getLogger(), LogLevel.DEBUG)
```

A test can log a message by calling the `Logger` method related to the relevant log level. Refer to the `Logger` documentation of the SLF4J documentation for details.

```
class MySampleServerTest extends PCServerTestClassBase {

  function testMySampleServerTest() {
    Logger.info("Running sample server test")       // Log a message at the INFO level

    // ... Perform test

    // ... Log messages at other levels
    Logger.debug("Debug information")               // Log a message at the DEBUG level
    Logger.error("Error information")               // Log a message at the ERROR level
    Logger.trace("Details about the test")          // Log a message at the TRACE level
    Logger.warn("Potential issue in the test")      // Log a message at the WARN level
  }
}
```

Method: startLogCapture

```
startLogCapture(logger : Logger) : CapturingLogger
```

The `startLogCapture` method creates a new logging object that captures subsequent logging events.
The method returns a `CapturingLogger` object.

```
uses gw.testharness.CapturingLogger
```

```
class MySampleServerTest extends PCServerTestClassBase {

  function testMySampleServerTest() {
    var logger : CapturingLogger

    try {
      logger = startLogCapture(getLogger())         // Start capturing the test logger

      assertThat(logger.getCapturedEvents())        // Verify that log is initially empty
            .as("Log should be initially empty")
            .isEmpty()

      // ... Perform test here

      assertThat(logger.getCapturedEvents())        // Verify that log is no longer empty
            .as("Should have logged messages")
            .isNotEmpty()
  }
}
```

Miscellaneous

The framework provides several general purpose functions.

Method: getName

```
getName() : String
```

The `getName` method retrieves the value of the object's `name` property.

The `name` property can be assigned in the object's constructor or by calling the `setName` method.

Method: getUniqueSuffixForTest

```
getUniqueSuffixForTest() : String
```

The `getUniqueSuffixForTest` method generates a globally unique string. Possible uses of the string include avoidance of duplicate keys and as a suffix to the test object's `name` property.

```
mySampleTest01.setName("MySampleTest" + mySampleTest01.getUniqueSuffixForTest())
mySampleTest02.setName("MySampleTest" + mySampleTest02.getUniqueSuffixForTest())
```

Method: setName

```
setName(name : String) : void
```

The `setName` method assigns the value of the `name` argument to the object's `name` property.

Method: toString

```
toString() : String
```

The `toString` method returns a string representation of the test object.

In the base class implementation, the object's string representation is constructed by concatenating the following values.

- The object's name property
- The object's package name
- The object's class name

The format of the concatenated string is `name(package.class)`.

Create a framework test suite

Multiple test classes can be combined to form a single test suite. Each test class in a suite must be based on the same class type. For example, `MyServerTestSuite` might consist of the test classes `MyServerTestOne`, `MyServerTestTwo`, and `MyServerTestThree`, where each class is based on `PCServerTestClassBase`.

A test suite is created by defining a suite class. The suite class must define a static function called `suite`. The `suite` function creates the test suite by using the `SuiteBuilder` class.

The `SuiteBuilder` class defines a constructor and the following methods.

- `withSuiteName`: Accepts a `String` suite name argument. The generated suite will include all the test classes defined with a @Suites annotation value that matches the suite name argument. To create a suite that includes all the test classes that were defined *without* a @Suites annotation, do not call the `withSuiteName` method when building the `SuiteBuilder`.
- `build`: Accepts no arguments. Creates the test suite object.

The following suite class builds a suite that includes each test class that specified a @Suites annotation with the `SAMPLE_SUITE_NAME` argument. Also, each test class in the suite extends the `PCUnitTestClassBase` class.

```
package gw.suites

uses gw.suites.MySuiteNames
uses gw.api.test.PCUnitTestClassBase
uses gw.api.test.SuiteBuilder
uses junit.framework.Test

class MySuiteClass {

  static function suite() : Test {
    return new SuiteBuilder(PCUnitTestClassBase)
      .withSuiteName(MySuiteNames.SAMPLE_SUITE_NAME)
      .build()
  }
}
```

The framework calls the `suite` method automatically during normal framework processing. The method returns a `JUnit`-based `Test` object. The `Test` object is managed by the framework and does not need to be manipulated by test code.

Run a test suite

To run a test suite, execute the following command line operations.

```
gwb compile
gwb runSuite -Dsuite=[TestSuiteClassName] [optional -D arguments]
```

The `gwb compile` command compiles the Gosu test and test suite classes.

The `gwb runSuite` command runs the tests in a specified test suite. The command accepts the following options. Each option is prefixed by `-D`.

Option	Description
`suite`	Required. Fully-qualified suite class name, such as `com.acme.MyTestSuite`. Example: `-Dsuite=com.acme.ux.tests.UxTestSuite`
`dir.results`	Directory to store results. If a relative directory is specified, the location is relative to the application's root directory. Optional. The results file contents are in a format that can be processed by continuous integration systems. Default: The `/tmp` directory. Example: `-Ddir.results=build/test-results`

Option	Description
`dir.temp`	Directory to store temporary test files. If a relative directory is specified, the location is relative to the application's root directory. Optional. Default: The `/tmp` directory. Example: `-Ddir.temp=build/test-temp`
`memory`	Maximum memory in MB to allocate to run tests. Optional. Default: 768 Example: `-Dmemory=2048`

chapter 31

Using entity builders to create test data

Note: Guidewire does not recommend or support the use of classes that extend `gw.api.databuilder.DataBuilder` or classes that reside in the `gw.api.databuilder.*` package in a production environment. Guidewire provides GUnit as a development test facility only.

As you run tests against code, you need to run these test in the context of a known set of data objects. This set of objects is generally known as a *test fixture*. You use Gosu entity builders to create the set of data objects to use in testing.

Guidewire provides a number of entity "builders" as utility classes to quickly and concisely create objects (entities) to use as test data. The PolicyCenter base configuration provides builders for the base entities (such as `PolicyBuilder`, for example). However, if desired, you can extend the base `DataBuilder` class to create new or extended entities. You can commit any test data that you create using builders to the test database using the `bundle.commit` method.

For example, the following builder creates a new `Person` object with a `FirstName` property set to "Sean" and a `LastName` property set to "Daniels". It also adds the new object to the default test bundle.

```
var myPerson = new PersonBuilder()
  .withFirstName("Sean")
  .withLastName("Daniels")
  .create()
```

For readability, Guidewire recommends that you place each configuration method call on an indented separate line starting with the dot. This makes code completion easier. It also makes it simpler to alter a line or paste a new line into the middle of the chain or to comment out a line.

Gosu builders extend from the base class `gw.api.databuilder.DataBuilder`. To view a list of valid builder types in Guidewire PolicyCenter, use the Studio code completion feature. Type `gw.api.databuilder.` in the Gosu editor and Studio displays the list of available builders.

Package completion

As you create an entity builder, you must either use the full package path, or add a `uses` statement at the beginning of the test file. However, in general, Guidewire recommends that you place the package path in a `uses` statement at the beginning of the file.

```
uses gw.api.databuilder.AccountBuilder

@gw.testharness.ServerTest
```

```
class MyTest extends TestBase {

  construct(testname : String) {
    super(testname)
  }
  ...
  function testSomething() {
    //perform some test
    var account = new AccountBuilder().create()
  }
  ...
}
```

Guidewire provides certain of the `Builder` classes in `gw.api.builder.*` and others in `gw.api.databuilder`. Verify the package path as you create new builders.

Creating an entity builder

To create a new entity builder of a particular type, use the following syntax:

```
new TypeOfBuilder()
```

This creates a new builder of the specified type, with the Builder class setting various default properties on the builder entity. Each entity builder provides different default property values depending on its particular implementation. For example, to create (or build) a default address, use the following:

```
var address = new AddressBuilder()
```

To set specific properties to specific values, use the property configuration methods. The following are the types of property configuration methods, each which serves a different purpose as indicated by the method's initial word:

Initial word	Indicates
on	A link to a parent. For example, `PolicyPeriod` is on an `Account`, so the method is `onAccount(Account account)`.
as	A property that holds only a single state. For example, `asSmallBusiness` or `asAgencyBill`.
with	The single element or property to be set. For example, the following sets a `FirstName` property: `withFirstName("Joe")`

Use a `DataBuilder.with(...)` configuration method to add a single property or value to a builder object. For example, the following Gosu code creates a new `Address` object and uses a number of `with(...)` methods to initialize properties on the new object. It then uses an `asType(...)` method to set the address type.

```
var address = new AddressBuilder()
  .withAddressLine1( streetNumber + " Main St." )
  .withAddressLine2( "Suite " + suiteNumber)
  .withCity( "San Mateo" )
  .withState( "CA" )
  .withPostalCode( postalCode )
  .asType(typeStr)
  ...
```

After you create a builder entity, you are responsible for writing that entity to the database as part of a transaction bundle. In most cases, you must use one of the builder `create` methods to add the entity to a bundle. Which `create` method one you choose depends on your purpose.

To complete the previous example, add a `create` method at the end.

```
var address = new AddressBuilder()
  .withAddressLine1( streetNumber + " Main St." )
```

```
  ...
  .create()
```

Builder create methods

The `DataBuilder` class provides the following `create` methods:

```
builderObject.create( bundle )
builderObject.create()
builderObject.createAndCommit()
```

The following list describes these `create` methods.

Method	Description
`create()`	Creates an instance of this builder's entity type in the default bundle. This method does not commit the bundle. Studio resets the default bundle before every test class and method.
`createAndCommit()`	Creates an instance of this builder's entity type in the default bundle, and performs a commit of that default bundle.
`create(bundle)`	Creates an instance of this builder's entity type, with values determined by prior calls to the entity. The *bundle* parameter sets the bundle to use while creating this builder instance.

The no-argument create method

The no-argument `create` method uses a default bundle that all the builders share. This is adequate for most test purposes. However, as all objects created this way share the same bundle, committing the bundle on just one of the created objects commits all of the objects to the database. This also makes them available to the PolicyCenter interface portion of a test. For example:

```
var address = new AddressBuilder()
  .withCity( "Springfield" )
  .asHomeAddress()
  .create()

 new PersonBuilder()
  .withFirstName("Sean")
  .withLastName("Daniels")
  .withPrimaryAddress(address)
  .create()

 address.Bundle.commit()
```

In this example, `Address` and `Person` share a bundle, so committing `address.Bundle` also stores `Person` in the database. If you do not need a reference to the `Person`, then you do not need to store it in a variable.

GUnit resets the default bundle before every test class and method.

The create and commit method

The `createAndCommit` method is similar to the `create` method in that it adds the entity to the default bundle. It then, however, commits that bundle to the database.

The create with bundle method

If you need to work with a specific bundle, use the `create(bundle)` method. Guidewire recommends that you use this method inside of a transaction block. A transaction block provides the following:

- It creates the bundle at the same time as it creates the new builder.
- It automatically commits the bundle as it exits.

The following example illustrates the use of a data builder inside a transaction block.

```
uses gw.transaction.Transaction
```

```
function myTest() {
 var person : Person

 Transaction.runWithNewBundle( \ bundle -> {
   person = new PersonBuilder()
            .withFirstName( "John" )
            .withLastName( "Doe" )
            .withPrimaryAddress( new AddressBuilder()
                    .withCity( "Springfield" )
                    .asHomeAddress() )
            .create( bundle )
 } )

 assertEquals( "Doe", person.LastName )

}
```

Notice the following about this example:

- The example declares the `person` variable outside the transaction block, making it accessible elsewhere in the method.
- The data builder uses an `AddressBuilder` object nested inside `PersonBuilder` to build the address.
- The `Transaction.runWithNewBundle` statement creates the bundle and automatically commits it after Gosu Runtime executes the supplied code block.

In summary, the `create(`*`bundle`*`)` method does not create a bundle. Rather, it uses the bundle passed into it. Guidewire recommends that you use this method inside a transaction block that both creates the bundle and commits it automatically.

If you do not use this method inside a transaction block that automatically commits a bundle, then you must commit the bundle yourself. To do so, add `bundle.commit` to your code.

Entity builder examples

The examples in this section illustrate ways to use builders to create sample data for GUnit tests.

Creating multiple objects from a single builder

The Builder class creates the builder object at the time of the `create` call. Therefore, you can use the same builder instance to generate multiple objects.

```
uses gw.api.databuilder.ClaimBuilder

 var activity1 : Activity
var activity2 : Activity

 gw.transaction.Transaction.runWithNewBundle( \ bundle -> {
  var activityBuilder = new gw.api.databuilder.ActivityBuilder()
    .onClaim(new ClaimBuilder().uiReadyAuto().create(bundle) )
    .withType(  "general" )
    .withPriority( "high" )
  activity1 = activityBuilder.withSubject( "this is test activity one" ).create( bundle )
  activity2 = activityBuilder.withSubject( "this is test activity two" ).create( bundle )
}, "su")
```

Nesting builders

It is possible to nest one builder inside of another by having a method on a builder that takes another builder as an argument. For example, suppose that you want to create an `Account` that has an `AccountLocation`. In this situation, you might want to do the following:

```
var account = new AccountBuilder()
  .withAccountLocation(new AccountLocationBuilder()).createAndCommit()
```

Overriding default builder properties

The following code samples illustrates multiple ways to create an Account object. The first code sample shows a simple test method and uses a transaction block. The `Transaction` object takes a block, which assigns the new account to the variable in the scope outside of the transaction.

```
function myTest(){
  var account : Account
  Transaction.runWithNewBundle( \ bundle -> {
          account = new AccountBuilder().create(bundle)
  })
}
```

There are generally two kinds of accounts: person and company. By default, `AccountBuilder` creates a person account. If you want a company account, then you need to assign a company contact as the account holder, as shown in the following code sample:

```
uses gw.api.builder.AccountBuilder
uses gw.api.builder.CompanyBuilder

 var account = new AccountBuilder(false)
  .withAccountHolderContact(new CompanyBuilder(42))
  .createAndCommit()
```

In this example, passing `false` to `AccountBuilder` tells it not to create a default account holder. Instead, you pass in your own account holder by calling `withAccountHolderContact`, which takes a `ContactBuilder`. In this case, `CompanyBuilder` suffices. The passed in number 42 seeds the default data with something unique (ideally) and identifiable.

The following example creates a company account and overrides some of the default values. Anywhere you see `code`, it means a seed value. (String variants derive from the given values.) It also illustrates how to nest the results of one builder inside another.

```
uses gw.api.builder.AccountBuilder
uses gw.api.builder.CompanyBuilder
uses gw.api.builder.AddressBuilder
uses gw.api.databuilder.OfficialIDBuilder

 var code = 48

 var address = new AddressBuilder()
  .withAddressLine1(code + " Main St.")
  .withAddressLine2("Suite " + code)
  .withCity("San Mateo")
  .withState("CA")
  .withPostalCode("94404-" + code)
  .asBusinessAddress()

 var company = new CompanyBuilder(code, false)
  .withCompanyName("This Company " + code)
  .withWorkPhone("650-555-" + code)
  .withAddress(address)
  .withOfficialID(new OfficialIDBuilder().withType("FEIN").withValue("11-222" + code))

 var account = new AccountBuilder(false)
  .withIndustryCode("1011", "SIC")
  .withAccountOrgType( "Corporation" )
  .withAccountHolderContact(company)
  .createAndCommit()
```

The following example takes the previous code and presents it as a single builder that takes other builders as arguments. While more compact, it also takes more planning and understanding of builders to create. Notice the successive levels of indenting used to signal the creation of a new (embedded) builder.

```
uses gw.api.builder.AccountBuilder
uses gw.api.databuilder.OfficialIDBuilder
uses gw.api.builder.AddressBuilder
uses gw.api.builder.CompanyBuilder
```

```
var account = new AccountBuilder(false)
 .withIndustryCode("1011", "SIC")
 .withAccountOrgType("Corporation")
 .withAccountHolderContact(new CompanyBuilder(code, false)
   .withCompanyName("This Company " + code)
   .withWorkPhone("650-555-" + code)
   .withAddress(new AddressBuilder()
     .withAddressLine1(code + " Main St.")
     .withAddressLine2("Suite " + code)
     .withCity("San Mateo")
     .withState("CA")
     .withPostalCode("94404-" + code)
     .asBusinessAddress()
   )
   .withOfficialID(new OfficialIDBuilder()
     .withType("FEIN")
     .withValue("11-222" + code))
 )
 .createAndCommit()
```

Creating new builders

If you need additional builder functionality than that provided by the PolicyCenter base configuration builders, you can do either of the following:

- Extend an existing builder class and add new builder methods to that class.
- Extend the base `DataBuilder` class and create a new builder class with its own set of builder methods.

You can also create a builder for a custom entity that you created by extending the `DataBuilder` class.

Extending an existing builder class

To extend an existing builder class, use the following syntax:

```
class MyExtendedBuilder extends SomeExistingBuilder {
  construct() {
    ...
  }
  ...
  function someNewFunction() : MyExtendedBuilder {
    ...
    return this
  }
  ...
}
```

The following `MyPersonBuilder` class extends the existing `PersonBuilder` class. The existing `PersonBuilder` class contains methods to set a home phone number and mobile phone number. The new extended class contains a method to set the person's alternative phone number. As there is no static field for the properties on a type, you must look up the property by name. Note that you must first define the new property in the data model.

```
uses gw.api.databuilder.PersonBuilder

class MyPersonBuilder extends PersonBuilder {

  construct() {
    super( true )
  }

function withAltPhone( testname : String ) : MyPersonBuilder {
    set(Person#AltPhone, testname)
    return this
  }

 }
```

The `PersonBuilder` class has two constructors. This code sample uses the one that takes a Boolean that means create this class `withDefaultOfficialID`.

Another more slightly complex example would be if you extended the `Person` object and added a new `PreferredName` property. In this case, you might want to extend the `PersonBuilder` class also and add a `withPreferredName` method to populate that field through a builder.

Extending the DataBuilder class

To extend the `DatabBuilder` class, use the following syntax:

```
class MyNewBuilder extends DataBuilder<BuilderEntity, BuilderType> {
   ...
}
```

The `DataBuilder` class takes the following parameters:

Parameter	Description
`BuilderEntity`	Type of entity created by the builder. The `create` method requires this parameter so that it can return a strongly-typed value and, so that other builder methods can declare strongly-typed parameters.
`BuilderType`	Type of the builder itself. The `with` methods require this on the `DataBuilder` class so that it can return a strongly-typed builder value (to facilitate the chaining of `with` methods).

If you choose to extend the `DataBuilder` class (`gw.api.databuilder.DataBuilder`), place your newly created builder class in the `gw.api.databuilder` package in the Studio **Tests** folder. Start any method that you define in your new builder with one of the following recommended words:

Initial word	Indicates
on	A link to a parent. For example, `PolicyPeriod` is on an `Account`, so the method is `onAccount(Account account)`.
as	A property that holds only a single state. For example, `asSmallBusiness` or `asAgencyBill`.
with	The single element or property to be set. For example, the following sets a `FirstName` property: `withFirstName("Joe")`

Your configuration methods can set properties by calling `DataBuilder.set` and `DataBuilder.addArrayElement`. You can provide property values as any of the following:

- Simple values.
- Beans to be used as subobjects.
- Other builders, which PolicyCenter uses to create subobjects if it calls your builder's `create` method.
- Instances of `gw.api.databuilder.ValueGenerator`. For example, an instance that generates a different value to satisfy uniqueness constraints for each instance constructed.

`DataBuilder.set` and `DataBuilder.addArrayElement` optionally accept an integer order argument that determines how PolicyCenter configures that property on the target object. (PolicyCenter processes properties in ascending order.) If you do not provide an order for a property, Studio uses `DataBuilder.DEFAULT_ORDER` as the order for that property. PolicyCenter processes properties with the same order value (for example, all those that do not have an order) in the order in which they are set on the builder.

In most cases, Guidewire recommends that you omit the order value as you are implement builder configuration methods. This enables callers of your builder to select the execution order through the order of the configuration method calls.

Constructors for builders can call `set`, and similar methods to set up default values. These are useful to satisfy `null` constraints so it is possible to commit built objects to the database. However, Guidewire generally recommends that you limit the number of defaults. This is so that you have the maximum control over the target object.

Other DataBuilder classes

The `gw.api.databuilder` package also includes `gw.api.databuilder.ValueGenerator`. You can use this class to generate a different value for each instance constructed to satisfy uniqueness constraints. The `databuilder` package includes `ValueGenerator` class variants for generating unique integers, strings, and typekeys.

- `gw.api.databuilder.SequentialIntegerGenerator`
- `gw.api.databuilder.SequentialStringGenerator`
- `gw.api.databuilder.SequentialTypeKeyGenerator`

Custom builder populators

Ideally, all building can be done through simple property setters, using the `DataBuilder.set` or `DataBuilder.addArrayElements` methods. However, you may want to define more complex logic, if these methods do not suffice. To achieve this, you can define a custom implementation of `gw.api.databuilder.populator.BeanPopulator` and pass it to `DataBuilder.addPopulator`. Guidewire provides an abstract implementation, `AbstractBeanPopulator`, to support short anonymous `BeanPopulator` objects.

The following example uses an anonymous subclass of `AbstractBeanPopulator` to call the `withCustomSetting` method. This code passes the group to the constructor, and the code inside of `execute` only accesses it through the `vals` argument. This allows the super-class to handle packaging details.

```
public MyEntityBuilder withCustomSetting(group : Group) {

  addPopulator(new AbstractBeanPopulator<MyEntity>(group) {

    function execute(e : MyEntity, vals : Object[]) {
      e.customGroupSet(vals[0] as Group)
    }
  })
  return this
}
```

The `AbstractBeanPopulator` class automatically converts builders to beans. That is, if you pass a builder to the constructor of `AbstractBeanPopulator`, it returns the bean that it builds in the `execute` method. The following example illustrates this.

```
public MyEntityBuilder withCustomSetting(groupBuilder : DataBuilder<Group, ?>) : MyEntityBuilder {

  addPopulator(new AbstractBeanPopulator<MyEntity>(groupBuilder) {

     function execute(e : MyEntity, vals : Object[]) {
      e.customGroupSet(vals[0] as Group)
     }
  })
  return this
}
```

Create and test a builder for a custom entity

About this task

It is possible to create a builder for a custom entity. For example, suppose that you want each PolicyCenter user to have an array of external credentials for automatic sign-on to linked external systems. To implement this requirement, you can create an array of `ExtCredential` on `User`, with each `ExtCredential` having the following parameters.

Parameter	Type
`ExtSystem`	Typekey
`UserName`	String

Parameter	Type
Password	String

After creating your custom entity and its builder class, you need to test it. To accomplish this, you need to do the following:

Task	Affected files
1. Create a custom ExtCredential array entity and extend the User entity to include it.	ExtCredential.eti User.etx
2. Create an ExtCrendentialBuilder by extending the DataBuilder class and adding withXXX methods to it.	ExtCredentialBuilder.gs
3. Create a test class to exercise and test your new builder.	ExtCredentialBuilderTest.gs

To create a new array ExtCredential custom entity, do the following tasks, as shown in the procedure.

- Add the ExtSystem typelist by using the **Typelist** editor in Guidewire Studio.
- Define the ExtCredential array entity in the ExtCredential, by using the editor in Guidewire Studio.
- Modify the array entity definition to include a foreign key to User in ExtCredential.
- Add an array field to the User entity in User.etx.

Procedure

1. Add the ExtSystem typelist.
   a. In Guidewire Studio, navigate to **configuration→config→Extensions→Typelist**.
   b. Right-click **Typelist**, and then click **New→Typelist**.
   c. In **Name**, type ExtSystem. Then, click **OK**.
   d. Add a few *external system* typecodes.
      Add SystemOne, SystemTwo, or similar items.
2. Create the ExtCredential entity type.
   a. In Guidewire Studio, navigate to **configuration→config→Extensions→Entity**.
   b. Right-click **Entity**, and then click **New→Entity**.
   c. In **Entity**, type ExtCredential. Then, click **OK**.
   d. Set the exportable attribute to true.
   e. Set the platform attribute to true.
   f. Add a typekey with the following attributes:

      name
      ExtSystem

      typelist
      ExtSystem

      nullok
      false

      desc
      Type of external system
   g. Add a column with the following attributes:

      name
      UserName

type

```
shorttext
```

nullok

```
false
```

h. Add a column with the following attributes:

name

```
Password
```

type

```
shorttext
```

nullok

```
false
```

i. Add a `foreignkey` with the following attributes:

name

```
UserID
```

fkentity

```
User
```

nullok

```
false
```

desc

```
FK back to User
```

3. Modify the `User` entity.
   a. In Guidewire Studio, in **configuration→config→Extensions→Entity**, double-click `User.etx`.
   b. Add an array with the following attributes:

name

```
ExtCredentialRetirable
```

type

```
ExtCredential
```

desc

```
An array of ExtCredential objects
```

arrayfield

```
UserID
```

exportable

```
false
```

4. Create an `ExtCredentialBuilder` class that extends the base `DataBuilder` class. Place this class in its own package under **configuration** and in the `gsrc` folder.
   a. In Guidewire Studio, navigate to **configuration→config→gsrc**.
   b. Right-click **gsrc**, and then click **New→Package**.
   c. In **Enter new package name**, type a name for the new package. Then, click **OK**.
   Type `AllMyClasses`.
   d. Right-click **AllMyClasses**, and then click **New→Gosu Class**.
   e. In **Name**, type `ExtCredentialBuilder`. Then, click **OK**.

f. In the editor, enter the following code:

```
package AllMyClasses

uses gw.api.databuilder.DataBuilder

class ExtCredentialBuilder extends DataBuilder<ExtCredential, ExtCredentialBuilder>  {

  construct() {
    super(ExtCredential)
  }

  function withType (  type: typekey.ExtSystem ) : ExtCredentialBuilder {
    set(ExtCredential.TypeInfo.getProperty( "ExtSystem" ), type)
    return this
  }

  function withUserName( somename : String ) : ExtCredentialBuilder {
    set(ExtCredential.TypeInfo.getProperty( "UserName" ), somename)
    return this
  }

  function withPassword( password : String ) : ExtCredentialBuilder {
     set(ExtCredential.TypeInfo.getProperty( "Password" ), password)
     return this
  }
}
```

Notice the following about this code sample.

- It includes a `uses ... DataBuilder` statement.
- It extends the `Databuilder` class, setting the *`BuilderType`* parameter to `ExtCredential` and the *`BuilderEntity`* parameter to `ExtCredentialBuilder`.
- It uses a constructor for the super class—`DataBuilder`—that requires the entity type to create.
- It implements multiple `withXXX` methods that populate an `ExtCredential` array object with the passed in values.

5. Create an `ExtCredentialBuilderTest` class that is a GUnit test that uses the `ExtCredentialBuilder` class to create test data. Place this class in its own package in the **Tests** folder.

```
package MyTests

uses AllMyClasses.ExtCredentialBuilder
uses gw.transaction.Transaction

@gw.testharness.ServerTest
class ExtCredentialBuilderTest extends gw.testharness.TestBase {

  static var credential : ExtCredential
  construct() {
  }

  function beforeClass () {
    Transaction.runWithNewBundle( \ bundle -> {
      credential = new ExtCredentialBuilder()
              .withType(  "SystemOne" )
              .withUserName( "Peter Rabbit" )
              .withPassword( "carrots" )
              .create( bundle )
      }
    )
  }

  function testUsername() {
    assertEquals("User names do not match.", credential.UserName, "Peter Rabbit")
  }

  function testPassword() {
    assertEquals("Passwords do not match.", credential.Password, "carrots")
  }
}
```

Notice the following about this code sample:

- It includes the `uses` statements for both `ExtCredentialBuilder` and `gw.transaction.Transaction`.
- It creates a static `credential` variable. As the code declares this variable outside of a method—as a class variable—it is available to all methods within the class. (GUnit maintains a single copy of this variable.) As you run a test, GUnit creates a single instance of the test class that each test method uses. Therefore, to preserve a variable value across multiple test methods, you must declare it as a static variable.
- It uses a `beforeClass` method to create the `ExtCredential` test data. This method calls `ExtCredentialBuilder` as part of a transaction block, which creates and commits the bundle automatically. GUnit calls the `beforeClass` method before it instantiates the test class for the first time. Thereafter, the test class uses the test data created by the `beforeClass` method. It is important to understand that GUnit does not drop the database between execution of each test method within a test class. However, if you run multiple test classes together (for example, by running all the test classes in a package), GUnit resets the database between execution of each test class.
- It defines several test methods, each of which starts with `test`, with each method including an `assertXXX` method to test the data.

Result

If you run the `ExtCredentialBuilderTest` class as defined, the GUnit tester displays green icons, indicating that the tests were successful.

part 5

Gosu style

chapter 32

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, understandable, and maintainable by other people.

General coding guidelines

Use the following recommendations to improve the clarity and readability of your Gosu code.

Omit semicolons

Omit semicolons as line terminators, because they are unnecessary in almost all cases. Gosu code looks cleaner without semicolons.

Semicolons are only needed if you need to separate multiple Gosu statements all written on a single line in a one-line statement list. This construction is generally not recommended, but is sometimes appropriate for simple statement lists declared in-line in 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 coercion to a type that is already included at the end of the Gosu statement.

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.

The Gosu `==` operator is equivalent to the object method `equals(obj1.equals(obj2))` but the `==` operator is null-safe. 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.

Consider the following Gosu code that uses the `equals` method:

```
( planName.equals( row.Name.text ) )
```

Using the == operator is more readable, as shown in the following line:

```
( 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.

Gosu prevents you from comparing incompatible array types for equality with the == and != operators. 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"}
```

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 Gosu compares the individual elements of the table:

```
new Number[] {1,2} == new Object[] {"1","2"}
```

WARNING Be careful if comparing arrays. Gosu performs recursive comparison of individual elements for compatible array types.

You can also compare objects by using the === and !== operators.

Capitalization conventions

The following table lists conventions for capitalization of various Gosu language elements:

Language element	Standard capitalization	Example
Gosu keywords	Always specify Gosu keywords correctly, typically lowercase.	`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`
Package names	Lowercase all letters in packages and subpackages	`com.mycompany.*`

Because Gosu is case-sensitive, you must access existing types exactly as they are declared, including correct capitalization. Capitalization in the middle of a name is also important.

Use the Gosu editor's code completion feature to enter the names of types and properties correctly. This ensures standard capitalization.

Some entity and typelist APIs are case insensitive if they take a `String` value for the name of an entity, a property, or a typecode. However, best practice is to pass the name exactly as declared.

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 property name, which is the interface to other code, is separated from the storage and retrieval of the value.

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. Gosu syntax supports creating the local variable and the public property on the same line by using the `as` keyword followed by the property name.

For example, in your Gosu classes that define class variables, use this variable declaration syntax:

```
private var _firstName : String as FirstName
```

This line declares a private variable called `_firstname`, which Gosu exposes as a public property called `FirstName`.

Do not create a public variable, like the following line:

```
public var FirstName : String  // Do not do this. Public variable scope is not Gosu standard style.
```

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 behavior is particularly valuable for `if` statements and similar Gosu structures. In the Gosu code that the `if` statement bounds, you do not need to cast by using an `as` *`TYPE`* expression to the subtype. Because Gosu confirms that the object has the more specific subtype, Gosu implicitly considers that variable's type to be the subtype, within that block of code.

The structure of this type inference looks like the following code:

```
var VARIABLE_NAME : TYPE_NAME

   // This comparison assumes SUBTYPE_NAME is a subtype of TYPE_NAME
if (VARIABLE_NAME typeis SUBTYPE_NAME) {

  // Use the VARIABLE_NAME as SUBTYPE_NAME in this block without casting
}
```

For example, the following example declares a variable as an `Object`, but downcasts the type to `String` 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 code works because the `typeis` inference is effective immediately and propagates to adjacent expressions.

Note that `length` is a property on `String`, not `Object`. By using `typeis` to downcast from `Object` to `String`, you do not need an additional cast around the variable `x`. 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 readable, concise Gosu code. Do not write Gosu code with unnecessary casts.

See also

- "Equality expressions" on page 81
- "Properties" on page 151
- "Basic type checking" on page 403