Gosu Reference Guide

Gosu Release 0.9.0-C

Guidewire

Product Name: Guidewire Gosu
Product Release: 0.9.0-C
Document Name: Gosu Reference Guide
Document Revision: 17-January-2012

Contents

About This Document

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

Intended Audience

This document is intended for the following readers:

- Programmers who implement and maintain Gosu code
- Implementation team members who configure any part of an application that invokes Gosu.

Downloads, Technical Questions, and Submitting Feedback

To download latest version of the Gosu language and the Gosu documentation, go to:

`http://gosu-lang.org`

To ask questions about Gosu or offer general feedback about Gosu, join and post to the Gosu language forum:

`http://groups.google.com/group/gosu-lang`

To file bug reports, please submit them to the Gosu language bug tracking system:

`http://code.google.com/p/gosu-lang/issues/list`

Conventions In This Document

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

chapter 1

Gosu Introduction

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

This topic includes:

Welcome to Gosu

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

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

- *enhancements*, which add functions and properties to other types, even Java types. Gosu includes built-in enhancements to common Java classes, some of which add features that are unavailable in Java (such as blocks).
- *generics*, which abstracts the behavior of a type to work with multiple types of objects. The Gosu generics implementation is 100% compatible with Java, and adds additional powerful improvements. See “Generics in Gosu” on page 32 for details.
- *native XML/XSD support*
- *native web service (SOAP) support*
- *an extensible type system*, which means that custom type loaders can dynamically inject types into the language. You can use these new types as native objects in Gosu. For example, custom type loaders dynamically add Gosu types for objects from XML schemas (XSDs) and from remote WS-I compliant web services (SOAP). Later versions of the Gosu community release will include more APIs and documentation about creating your own type loaders.
- *large companies around the world use Gosu every day in production systems for critical systems.*

Basic Gosu

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

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

Gosu uses the Java type `java.util.String` as its native `String` type to manipulate texts. You can create in-line `String` literals just as in Java. In addition, Gosu supports native in-line templates, which simplifies common text substitution coding patterns. For more information, see “Gosu Templates” on page 26.

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

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

Despite appearances in this example, Gosu is *statically typed*. All variables have a compile-time type that Gosu enforces at compile time, even though in this example there is no *explicit* type declaration. In this example, Gosu automatically assigns these variables the type `int`. Gosu *infers* the type `int` from the expressions on the right side of the equals signs on lines that declare the variable. This language feature is called *type inference*. For more information about type inference, see “Type Inference” on page 30.

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

In Gosu, this looks like:

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

This is equivalent to the following Java code:

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

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

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

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

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

> **Note:** For more information, see “More About the Gosu Type System” on page 29 and “Gosu Classes and Properties” on page 18.

From the previous examples, you might notice another difference between Gosu and Java: no semicolons or other line ending characters. Semicolons are unnecessary in nearly every case, and the standard style is to omit them. For details, see “Gosu Statement Terminators” on page 35.

Control Flow

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
var x = "b"

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

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

Blocks

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

To define a block

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

For more information about blocks, see "Gosu Blocks" on page 165.

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

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

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

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

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

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

For more information about blocks, see "Gosu Blocks" on page 165. For more information about collections enhancement methods, many of which use blocks, see "Collections" on page 183.

Special Block Shortcut for One-Method Interfaces

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

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

For more information, see "Gosu Block Shortcut for Anonymous Inner Classes Implementing an Interface" on page 143.

Enhancements

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

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

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

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

For more information, see "Enhancements" on page 161.

Collections

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

In addition, Gosu adds the following features:

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

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

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

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

- Gosu includes built-in enhancements that improve Java collection classes. Some enhancements enable you to use Gosu features that are unavailable in Java. For example, the following Gosu code initializes a list of String objects and then uses enhancement methods that use Gosu blocks, which are in-line functions. (See "Blocks" on page 16).

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

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

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

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

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

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

For more information, see "Collections" on page 17.

Access to Java Types

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

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

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

For example:

- Gosu can instantiate Java types
- Gosu can manipulate Java objects (and primitives) as native Gosu objects.
- Gosu can get variables from Java types
- Gosu can call methods on Java types. For methods that look like getters and setters, Gosu exposes methods instead as properties.
- Gosu extends and improves many common Java types using Gosu *enhancements*. (See “Enhancements” on page 17.)
- You can also extend Java types and implement Java interfaces.

For more information, see “Java and Gosu” on page 235.

Gosu Classes and Properties

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

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

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

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

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

Note: You can optionally specify that your class implements interfaces. See “Interfaces” on page 22.

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

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

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

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

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

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

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

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

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

Unlike Java, you can omit the type name entirely in a `new` expression if the type is known from its context. For example:

```
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()    // notice the type name is omitted
t = new()    // notice the type name is omitted

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

// optionally OMIT 'new' keyword and still use the Gosu initialization syntax
t.Student = { :Name = "Bob Smith", :Age = 30 }
```

For more details, see “Creating and Instantiating Classes” on page 128 and “New Object Expressions” on page 86.

Functions

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

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

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

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

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

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

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

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

Gosu supports access control modifiers (`public`, `private`, `internal`, and `protected`) and they have the same meaning as in Java. For example, if you mark a method `public`, any other code can call that method. For more information, see “Access Modifiers” on page 136.

Gosu also supports static methods, which means methods on the type rather than on object instances. See “Static Members” on page 21.

If the return type is not `void`, **all** possible code paths must return a value in a method that declares a return type. In other words, if any code path contains a `return` statement, Gosu requires a `return` statement for all possible paths through the function. The set of all paths includes all outcomes of conditional execution, such as `if` and `switch` statements. This is identical to the analogous requirement in Java. The Gosu editor automatically notifies you at compile time of this issue if it happens. For details, see “Gosu Functions” on page 106.

Class Variables and Properties

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

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

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

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

In addition, Gosu has a special null-safety behavior with pure property paths, which are the form `obj.Property1.Property2.Property3`. For more information, see "Property Accessor Paths are Null Safe" on page 21.

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

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

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

The following is a simple Gosu class definition:

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

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

 class Person {

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

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

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

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

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

This allows you to use concise code like the following:

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

From Gosu, Java Get and Set Methods Become Properties

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

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

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

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

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

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

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

For a detailed example, see “Java Get and Set Methods Convert to Gosu Properties” on page 237.

Property Accessor Paths are Null Safe

For normal property access with the period character, all objects to the left of the period must be non-null at run time or Gosu throws an exception. For example:

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

Gosu provides a way to access properties in a way that is tolerant of unexpected `null` values, a feature called *null safety.* To do this, add a question mark before the period to transform the operator into the null-safe version. For example:

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

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

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

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

For more about Gosu null safety, see “Null Safety for Properties Other Operators” on page 30.

Static Members

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

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

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

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

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

To declare a type as static for a new Gosu class, use the `static` keyword just as in Java. For details, see "Modifiers" on page 135.

Interfaces

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

```
package example

 interface ILoadable {
    function load()
 }
```

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

```
package example

class LoadableThing implements ILoadable {

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

For more information, see "Interfaces" on page 22.

List and Array Expansion Operator *.

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

```
names*.Length
```

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

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

Gosu infers the type of the list as appropriate parameterized type using Gosu generics, an advanced type feature. For more information about generics, see "Generics in Gosu" on page 32. Similarly, Gosu infers the type of the result array if you originally call the operator on an array.

This feature also works with both arrays and lists. For detailed code examples, see "List and Array Expansion (*.)" on page 187.

Comparisons

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

- The operators `>`, `<`, `>=`, and `<=` operators work with all objects that implement the `Comparable` interface, not just numbers.
- The standard equal comparison `==` operator implicitly uses the `equals` method on the first (leftmost) object. This operator does not check for pointer equality. It is `null` safe in the sense that if either side of the operator is null, Gosu does not throw a null pointer exception. (For related information, see "Property Accessor Paths are Null Safe" on page 21.)

  Note: In contrast, in the Java language, the `==` operator evaluates to `true` if and only if both operands have the same exact **reference value**. In other words, it evaluates to `true` if they refer to the same object in memory. This works well for primitive types like integers. For reference types, this usually is not what you want to compare. Instead, to compare *value equality*, Java code typically uses `*object*.equals()`, not the `==` operator.
- There are cases in which you want to use identity reference, not simply comparing the values using the underlying `*object*.equals()` comparison. In other words, some times you want to know if two objects literally reference the same in-memory object. Gosu provides a special equality operator called `===` (three equals signs) to compare object equality. It always compares whether both references point to the same in-memory object. The following examples illustrate some differences between `==` and `===` operators:

Expression	Prints this Result	Description

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

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

Case Sensitivity

Gosu language itself is case insensitive, but Gosu compiles and runs faster if you write all Gosu as case-sensitive code matching the declaration of the language element. Additionally, proper capitalization makes your Gosu code easier to read. For more information, including Gosu standards for capitalizing your own language elements, see "Gosu Case Sensitivity" on page 33.

Compound Assignment Statements

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

- `++` , which is the increment-by-one operator, supported only after the variable name
- `+=` , which is the add-and-assign operator, supported only after the variable name followed by a value to add to the variable
- Similarly, Gosu supports `--` (decrement-by-one) and `-=` (subtract-and-assign)

- Gosu supports additional compound assignment statements that mirror other common operators. See “Variable Assignment” on page 98 for the full list.

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

```
i++
```

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

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

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

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

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

See “Variable Assignment” on page 98 for more details.

Delegating Interface Implementation with Composition

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

```
package test

class MyWindow implements IClipboardPart {
  delegate _clipboardPart represents IClipboardPart

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

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

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

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

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

For more information, see “Composition” on page 151.

Concurrency

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

Gosu provides the following concurrency APIs:

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

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

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

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

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

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

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

For more information about concurrency APIs, see “Concurrency” on page 269

Exceptions

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

Catching Exceptions

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

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

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

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

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

For example:

```
try {
  doSomethingThatMayThrowIOException()
}
```

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

Throwing Exceptions

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

The following example creates an explicit `RuntimeException` exception:

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

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

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

Annotations

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

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

```
class MyClass{

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

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

You can get annotations from types at run time.

Gosu supports named arguments syntax for annotations:

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

For more information, se “Annotations” on page 155.

Gosu Templates

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

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

If you print this variable, Gosu outputs:

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

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

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

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

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

For more information, see “Gosu Templates” on page 243.

Native XML and XSD Support

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

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

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 manipulates XML objects using XSD-based types:

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

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

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

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

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

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

For more information, see “Gosu and XML” on page 195.

Native Web Service Support Using a WSDL Type Loader

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

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

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

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

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

Gosu Character Set

Because Gosu runs within a Java Virtual Machine (JVM), Gosu shares the same 16-bit Unicode character set as Java. This allows you to represent a character in virtually any human language in Gosu.

Running Gosu Programs and Calling Other Classes

To use Gosu, the initial file that you run **must** be a Gosu program. A Gosu program file has the `.gsp` file name extension. Gosu code in a program can call out to other Gosu classes and other types. For more information about Gosu programs, see “The Structure of a Gosu Program” on page 59.

You can run Gosu programs (`.gsp` files) directly from the command line or from within an IDE such as IntelliJ. You cannot run a Gosu class file or other types file directly from within an IDE such as IntelliJ. If you want to call a Gosu class (or other type of file), make a simple Gosu program that uses your other types.

In Java, you would define a `main()` method in a class and tell Java which main class to run. It would call out to other classes as needed.

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

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

- Use the `classpath` **argument** on the command line tool. See “Command Line Tool Options” on page 58.
- Add a `classpath` **statement** at the top of your Gosu program.

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

1. Create a package-style hierarchy for your class files somewhere on your disk. For example, if the root of your files are `Gosu/MyProject/`, put the class files for the Gosu class `com.example.MyClass` at the location `Gosu/MyProject/com/example/MyClass.gs`.
2. In your Gosu program, tell Gosu where to find your other Gosu classes and Java classes by adding the class-path statement.

Typically you would place Java classes, Gosu classes, or libraries in subdirectories of your main Gosu program.

For example, suppose you have a Gosu program at this location:

```
C:\gosu\myprograms\test1\test.gsp
```

Copy your class file for the class `mypackage.MyClass` to the location:

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

Copy your library files to locations such as:

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

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

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

For more details, see “The Structure of a Gosu Program” on page 59.

More About the Gosu Type System

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

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

- Compile Time Error Prevention
- Intelligent Code Completion and Other Gosu Editor Tools
- Type Usage Searching

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

- “Type System” on page 251
- “Basic Type Coercion” on page 251
- “Variable Type Declaration” on page 98.

Although Gosu is a statically-typed language, Gosu supports a concept of generic types, called *Gosu generics*. You can use generics in special cases to define a class or method so that it works with multiple types of objects. Gosu generics are especially useful to design or use APIs that manipulate *collections* of objects. For a summary, see “Generics in Gosu” on page 32, or the full topic “Gosu Generics” on page 173. Programmers familiar with the Java implementation of generics quickly become comfortable with the Gosu implementation of generics.

Compile Time Error Prevention

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

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

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

For example, consider the following function definition.

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

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

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

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

Type Inference

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

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

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

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

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

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

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

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

Intelligent Code Completion and Other Gosu Editor Tools

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

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

Type Usage Searching

Complete Gosu plugins for IDEs (such as IntelliJ IDEA) support search for all occurrences of the usage of an object of a particular type. This is more than just textual search, but semantic search. See "Getting Started with Gosu Community Release" on page 43.

Null Safety for Properties Other Operators

In Gosu, a period character gets a property from an object or calls a method.

By default, the period operator is not null-safe. This means that if the value on the left side of the period evaluates to `null` at runtime, Gosu throws a *null pointer exception* (NPE). For example, `obj.PropertyA.PropertyB` throws an exception if `obj` or `obj.PropertyA` are null at run time.

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

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

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

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

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

For example, suppose you had a variable called `house`, which contained a property called `Walls`, and that object had a property called `Windows`. You could get the `Windows` value with the following syntax:

```
house.Walls.Windows
```

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

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

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

The following concise Gosu code is equivalent to the previous example and avoids any null pointer exceptions:

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

Null Safe Method Calls

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

For example:

```
house.myaction()
```

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

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

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

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

Null-Safe Versions of Other Operators

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

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

```
var displayName = Book.Title ?: "(Unknown Title)" // return "(Unknown Title)" if Book.Title is null
```

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

```
var book = bookshelf?[bookNumber] // return null if bookshelf is null
```

- The null-safe math operators (`?+`, `?-`, `?*`, `?/`, and `?%`). For example:

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

See "Handling Null Values In Expressions" on page 95.

Design Code for Null Safety

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

You can also design your code to take advantage of this special language feature. For example, expose data as *properties* in Gosu classes and interfaces rather setter and getter methods. This allows you to use the null-safe property operator (the `?.` operator), which can make your code both powerful and concise.

See Also

- For more examples and discussion, see "Handling Null Values In Expressions" on page 95

IMPORTANT For more information about property accessor paths and designing your APIs around this feature, see "Handling Null Values In Expressions" on page 95.

Generics in Gosu

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

```
uses java.util.*

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

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

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

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

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

  In Java, this is a compilation error:

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

  The analogous Gosu code works:

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

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

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

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

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

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

For more information, see "Gosu Generics" on page 173.

Gosu Primitives Types

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

Additionally, every Gosu primitive type (other than the special value `null`) has an equivalent object type defined in Java. For example, for `int` there is the `java.lang.Integer` type that descends from the `Object` class. This category of object types that represent the equivalent of primitive types are called *boxed primitive* types. In contrast, primitive types are also called *unboxed primitives*. In most cases, Gosu converts between boxed and unboxed primitive as needed for typical use. However, they are slightly different types, just as in Java, and on rare occasion these differences are important. Refer to "Type Object Properties" on page 258 for details.

Gosu Type Loaders

The Gosu type system has an open type system. An important part of this is that Gosu supports custom type loaders. A type loader dynamically injects types into the language and attaches potentially complex dynamic behaviors to working with the type. For example, a custom type loader adds types to the type system. Each time Gosu code accesses a property or calls methods on the custom types, you can run custom code. Gosu calls custom type loader code each time any Gosu code accesses a property or calls methods on the custom types.

There are several built-in type loaders:

- **Gosu XML/XSD type loader.** This type loader supports the native Gosu APIs for XML. For more information, see "Gosu and XML" on page 195.
- **Gosu SOAP/WSDL type loader.** This type loader supports the native Gosu APIs for the web services SOAP protocol. This works through a Gosu type loaders that reads web service WSDL files and lets you interact with the external service through a natural syntax and type-safe coding. For more information, see "Consuming WS-I Web Service Overview" on page 225
- **Property file type loader.** This type loader finds property files in the hierarchy of files on the disk along with your Gosu class files. Gosu creates types in the appropriate package (by the property file location) for each property. Access the properties directly in Gosu in a type-safe manner. For more information, see "Properties Files" on page 277.

You do not need to do anything special to install or enable these type loaders. Gosu includes these type loaders automatically for all Gosu code.

A future Gosu release will include documentation and supported APIs for creating custom type loaders.

IMPORTANT Later versions of the Gosu community release will include more APIs and documentation about creating your own custom Gosu type loaders.

Gosu Case Sensitivity

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

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

Language element	Standard capitalization	Example
Gosu keywords	Always specify Gosu keywords correctly as they are declared, typically lowercase. Java keywords are case-sensitive.	`if`
type names	uppercase first character	`DateUtil` `Claim`
variable names	lowercase first character	`myClaim`
method names	lowercase first character	`printReport`
property names	uppercase first character	`Name`
package names (case sensitive)	lowercase entire package name when creating new packages Always specify package names correctly as they are declared. Package names are case sensitive.	`com.mycompany.*`
Java types (case sensitive)	Java types require case sensitivity Always specify Java types correctly as they are declared. Java type names are case-sensitive.	`java.util.String`

Remember to access these items exactly as they are declared.

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

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

Instead, use the code:

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

Similarly, use class names properly. Do not write:

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

Instead, use the code:

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

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

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

Instead, use the code

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

It is best to change any existing code to be case sensitive, and write any new code to follow these guidelines.

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

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

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

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

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

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

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

Use the Gosu editor Code Completion feature to enter the names of types and properties correctly. This ensures standard capitalization.

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

Gosu Statement Terminators

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

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

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

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

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

For typical code, omit semicolons as they are unnecessary in almost all cases. It is standard Gosu style to use semicolons between multiple Gosu statements when they are all on one line. For example, as in a short Gosu *block* definition (see "Gosu Blocks" on page 165). However, even in those case semicolons are optional in Gosu.

Valid and Recommended

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

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

Valid But Not Recommended

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

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

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

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

Invalid Statements

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

Gosu Comments

Comment your Gosu code as you write it. The following table lists the comment styles that Gosu supports.

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

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

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

Gosu Reserved Words

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

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

- `hide`
- `implements`
- `index`
- `interface`
- `internal`
- `native`
- `unless`
- `uses`
- `var`
- `void`
- `while`

Notable Differences Between Gosu and Java

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

Difference	Java	Gosu	Required change?
General Differences			
Gosu language itself is case insensitive, but Gosu compiles and runs faster if you write Gosu as case-sensitive code. Match the declaration of each language element. See "Case Sensitivity" on page 23.	a.B = c.D B and D must exactly match the field declarations.	a.B = c.D Match the code capitalization to match the property declarations. Other capitalizations work, but are not recommended, such as: `a.b = c.d`	Optional
Omit semicolons in most code. Gosu supports the semicolon, but standard coding style is to omit it. (one exception is in block declarations with multiple statements)	`x = 1;`	`x = 1`	Optional

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

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

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

<table>
<tr><th>Difference</th><th>Java</th><th>Gosu</th><th>Required change?</th></tr>
<tr><td>In variable declarations, use the <code>var</code> 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</td><td><code>Auto c = new Auto()</code></td><td>Type inference
<code>var c = new Auto()</code>
Explicit:
<code>var c : Auto = new Auto()</code>
Type inference with coercion:
<code>var c = new Auto() as Vehicle</code></td><td>Required</td></tr>
<tr><td>To declare variable argument functions, also called <code>vararg</code> functions, Gosu does not support the special Java syntax. In other words, Gosu does not support arguments with “<code>...</code>” declarations, which indicates variable number of arguments. Instead, design APIs to use arrays or lists.

To call variable argument functions, pass an array of the declared type. Internally, in Java, these variable arguments are arrays. Gosu array initialization syntax is useful for calling these types of methods.</td><td><code>public String format(Object... args);</code></td><td><code>// function declaration</code>
<code>public function format(args : Object[])</code>

<code>// method call using</code>
<code>// initializer syntax</code>
<code>var c = format({"aa","bb"})</code></td><td>Required</td></tr>
<tr><td>Gosu supports the unary operator assignment statements <code>++</code> and <code>--</code>. However:
• only use the operator after the variable (such as <code>i++</code>)
• these only form statements not expressions.

There are other supported compound assignment statements, such as <code>+=</code>, <code>-=</code>, and others. see “Variable Assignment” on page 98.</td><td><code>if (++i > 2) {</code>
<code> //</code>
<code>}</code></td><td><code>i++</code>
<code>if (i > 2) {</code>
<code> //</code>
<code>}</code></td><td>Required</td></tr>
<tr><td>For static members (static methods and static properties), in Gosu you must qualify the type on which the static member appears. Use the period character to access the member. The type does not need to be fully qualified, though.</td><td><code>cos(PI * 0.5)</code></td><td><code>Math.cos(Math.PI * 0.5)</code>
Note that you do not need to fully qualify the type as <code>java.lang.Math</code>.</td><td>Required if you omit type names in your Java code before static members</td></tr>
<tr><td colspan="4">Type System</td></tr>
<tr><td>For coercions, use the <code>as</code> keyword.

Optionally, Gosu supports Java-style coercion syntax for compatability.</td><td><code>int x = (int) 2.1</code></td><td><code>// Gosu style</code>
<code>var x = 2.1 as int</code>

<code>//Java compatability style</code>
<code>var x = (int) 2.1</code></td><td>Optional</td></tr>
<tr><td>Check if an object is a specific type or its subtypes using <code>typeis</code>. This is similar to the Java <code>instanceof</code> operator.</td><td><code>myobj instanceof String</code></td><td><code>myobj typeis String</code></td><td>Required</td></tr>
</table>

Difference	Java	Gosu	Required change?

Gosu automatically downcasts to a more specific type in `if` and `switch` statements. Omit casting to the specific type. See “Automatic Downcasting for ‘typeis’ and ‘typeof’” on page 254.	`Object x = "nice"`
	`Int sl = 0`
	`if( x instanceof String ) {`
	`  sl = ((String) x).length`
	`}`
		`var x : Object = "nice"`
		`var sl = 0`
		`if( x typeis String ) {`
		`  sl = x.length // downcast`
		`}`
			Optional

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

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

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

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

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

Control Flow			

Difference	Java	Gosu	Required change?

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

The `for` loop syntax in Gosu is different for iterating a loop an integer number of times. The loop variable contains the a zero-based index.
 	
Gosu has native support for *intervals*, which are sequences of values of the same type between a given pair of endpoint values. For instance, the set of integers beginning with 0 and ending with 10 is the shorthand syntax `0..10`. Intervals are particularly useful to write concise easy-to-understand `for` loops.
 	
Gosu does not support the for(*initialize;compare;increment*) syntax in Java. However, you can duplicate it using a `while` statement (see example).
	`for(int i=1; i<20; i++){`
	`  //`
	`}`
		`for (item in 20) {`
		`  //`
		`}`
		Using Gosu intervals:
		`for( i in 1..50 ) {`
		`  print( i )`
		`}`
		verbose style:
		`var i = 0`
		`while (i < 20) {`
		`  //`
		`  i++`
		`}`
			Required

Other Gosu-specific features			

Gosu enhancements, which allow you to add additional methods and properties to any type, even Java types. See "Enhancements" on page 161.	*n/a*	`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. They are especially useful with the Gosu collections enhancements See "Gosu Blocks" on page 165. Blocks can also be useful as a shortcut for implementing one-method interfaces (see "Blocks as Shortcuts for Anonymous Classes" on page 171).	*n/a*	`\ x : Number -> x * x`	Optional
Native XML support and XSD support. See "Gosu and XML" on page 195.	*n/a*	`var e = schema.parse(xmlText)`	Optional

Difference	Java	Gosu	Required change?
Native support for consuming web services with syntax similar to native method calls. See “Consuming WS-I Web Service Overview” on page 225.	*n/a*	`extAPI.myMethod(1, true, "c")`	Optional
Native `String` templates and file-based templates with type-safe parameters. See “Gosu Templates” on page 243.	*n/a*	`var s = "Total = ${ x }."`	Optional

Gosu uses the Java-based collections APIs but improves upon them:
• Simplified initialization syntax that still preserves type safety.
• Simple array-like syntax for getting and setting values in lists, maps, and sets
• Gosu adds new methods and properties to improve functionality of the Java classes. Some enhancements use Gosu blocks for concise flexible code.
For new code, use the Gosu style initialization and APIs. However, you can call the more verbose Java style for compatibility. See “Collections” on page 183.
		`// easy initialization`
		`var strs = {"a", "ab", "abc"}`
		
		`// array-like "set" and "get"`
		`strs[0] = "b"`
		`var val = strs[1]`
		
		`// new APIs on Java`
		`// collections types`
		`strList.each(\ str -> { print(str) } )`
			Optional

List and array expansion operator. See “List and Array Expansion (*.)” on page 187.	*n/a*	`names*.Length`	Optional

Get Ready for Gosu

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

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

The next step for you is to write your first Gosu program and become familiar with the Gosu editor, either the built-in one or the plugin for JetBrains IntelliJ IDEA.

For the latest version of the Gosu language, the Gosu documentation, and information about IDE editors, refer to:

```
http://gosu-lang.org
```

To ask questions about Gosu or offer general feedback on the Gosu language, join and post to the gosu-lang forum:

```
http://groups.google.com/group/gosu-lang
```

To file bug reports, please submit them to the gosu-lang bug tracking system:

```
http://code.google.com/p/gosu-lang/issues/list
```

To continue your introduction to the Gosu language, see the following topic: “Getting Started with Gosu Community Release” on page 43.

chapter 2

Getting Started with Gosu Community Release

System Requirements

The following table lists the system requirements for the community release of Gosu.

Requirement	Supported versions	Required?
Java language	Java version 1.5 (Sun/Oracle J2SE release 5) or later on your computer. Gosu works with both the JRE version and the SDK version of Java. For Java downloads, visit `http://java.com`	Yes
Operating system	Any operating system that supports Sun Java version 1.5 or later.	Yes
JetBrains IntelliJ IDEA IDE	Version 11.0.x.	Required only for use with the IntelliJ IDEA IDE. You do not need an IDE to run Gosu programs from the command line or to use the self-contained visual Gosu editor. For more information about the command line tool, which includes the editor, see “Gosu Programs and Command Line Tools” on page 57.

Gosu and IntelliJ IDEA

There is a plugin for the IntelliJ IDEA IDE that enables editing, debugging, and running Gosu within the IDE. It features Gosu code completion, auto-detection of changed dependent files, semantic searching, and refactoring tools. For more information about providing feedback to the Gosu team about this plugin, see “Get Ready for Gosu” on page 41.

The Gosu plugin appears in the list of plugins within the IntelliJ IDEA application itself. You can install the Gosu plugin from within that user interface.

For installation instructions and full feature list for the Gosu plugin for the IntelliJ IDEA IDE, visit:

```
http://gosu-lang.org
```

To run Gosu programs *outside* the IntelliJ IDE, for example from the command line, download the official Gosu install available at the same URL. That download includes the latest Gosu language documentation.

Refer to the IntelliJ documentation for complete information about:

- creating new class files
- organizing modules
- refactoring tools
- file management (including Java JAR files)

Installing Gosu as Command Line Tool

To use Gosu as a command line tool, run the appropriate executable in the `bin` directory in the Gosu download.

To use Gosu as a Windows command line tool

1. Download the `gosu.zip` file from the location:

   ```
   http://gosu-lang.org/gosu/downloads/gosu.zip
   ```

2. Unzip the file to the desired location on your local disk. The documentation refers to this root directory for gosu simply as the name `Gosu`. The `Gosu/bin` directory contains the binary executables and the `Gosu/lib` directory contains the Gosu libraries.

1. Open a DOS command line window (`cmd.exe`)

2. Type the command:

   ```
   Gosu/bin/gosu.bat
   ```

3. When the Gosu prompt ("`gs>`") appears, type

   ```
   print("hello world!")
   ```

   The console prints the following

   ```
   hello world!
   ```

4. Experiment by typing other Gosu expressions or programs.

5. When you are done, type the one-word command `quit`, and then press the `Enter` key.

6. Optionally, you may want to add the `Gosu/bin` directory to your system path. That allows you to type `gosu` at command line, independent of the current working directory. Go to the Start menu and choose Control Panel → System → Environment Variables. Next, choose the `Path` variable. At end of the current path, append a semicolon and the full path to the `bin` directory. For example, suppose you installed (copied) the Gosu shell directory to the path:

   ```
   C:\gosu\
   ```

   Add the following to the system path:

   ```
   ;C:\gosu\bin
   ```

   To test this, close any existing command prompt windows, then open a new command prompt window and type the word `gosu` then the type the `Enter` key.

To use Gosu as a Unix, Linux, or Mac OS X command line tool

1. Open a Terminal window

2. Run the file:

```
Gosu/bin/gosu.sh
```

3. When the Gosu prompt ("gs>") appears, type

```
print("hello world!")
```

The console prints the following

```
hello world!
```

4. Experiment by typing other Gosu expressions or programs.
5. When you are done, type the one-word command `quit`, and then press the `Enter` key.
6. For more information about command line options, see “Gosu Programs and Command Line Tools” on page 57.

Advanced Examples

Servlet Example

The Gosu community release includes an example called `Servlet`. It is a simple implementation of a web server servlet implemented in Gosu.

To use the Gosu servlet example:

1. Install the Apache Tomcat web server. This example was written for Tomcat 6.0, which you can download at:

```
http://tomcat.apache.org/download-60.cgi
```

2. Set the system environment variable `CATALINA_HOME` to the root directory of your Tomcat installation. In Windows, go to Control Panel → System → Advanced → Set Environment Variables. In the bottom pane, click New and then create a new variable called `CATALINA_HOME` with value `C:\Program Files\apache-tomcat`, or wherever your installation is.
3. Create a servlet folder called `GosuServlet` in your web server’s `webapps` directory. For this example, we assume you are using Apache Tomcat and the directory is called `apache-tomcat`. Copy the GosuServlet files

```
- apache-tomcat
    - webapps
      - GosuServlet
```

4. Add a `WEB-INF` folder under your new servlet directory:

```
- GosuServlet
    - WEB-INF
```

5. Create a `web.xml` file inside that folder to enable Gosu template support, such as

```
- GosuServlet
    - WEB-INF
      - web.xml
```

In this file paste the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
       "http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

    <display-name>Gosu Servlet Examples</display-name>
    <description>
      Gosu Servlet Examples.
    </description>

    <servlet>
      <servlet-name>GosuServlet</servlet-name>
      <servlet-class>gw.util.servlet.GosuServlet</servlet-class>
    </servlet>
    <servlet-mapping>
      <servlet-name>GosuServlet</servlet-name>
```

```
      <url-pattern>*.gst</url-pattern>
    </servlet-mapping>
  </web-app>
```

6. Copy `HelloWorld.gst` from the example `example/GosuServlet/src/sample` directory into your servlet root directory. After this step, your file hierarchy looks like:

```
- apache-tomcat
  - webapps
    - GosuServlet
      - sample
        - hello
           - HelloWorld.gst
```

7. Copy **all** JAR files from the root of the servlet example directory into your servlet `WEB-INF` directory. Next, copy the JAR files from the `lib` subdirectory of the servlet example into your servlet `WEB-INF` directory. Your servlet looks like this:

```
- apache-tomcat
  - webapps
    - GosuServlet
      - WEB-INF
        - lib
          - gw-gosu-core.jar
          - gw-gosu-core-api.jar
          - servlet-api
          ...[the rest of the JAR files]
```

8. Start up Apache Tomcat. You can do this by running the following script:

```
apache-tomcat/bin/startup.bat
```

9. Wait for Tomcat to startup. If you see any servlet startup errors in the console, make a note of them for debugging or bug reporting.

10. Test this in your browser, remembering to set the domain name and port appropriately, such as:

```
http://localhost:8080/GosuServlet/sample/hello/HelloWorld.gst
```

Hibernate Database Example

The `examples` directory in the Gosu directory includes an example called `hibernate`. This shows an example project that uses the Hibernate tool. Hibernate is a Java library that provides a framework for mapping an object-oriented domain model to a traditional relational database. This is also called an object-relational mapping (ORM). Hibernate is free and open source software.

Gosu can easily access any Java classes or JAR files, so Gosu can access these valuable persistence libraries.

For more information about the Hibernate project, refer to:

```
http://www.hibernate.org/
```

The example contains a couple Gosu files and a directory of libraries (including the Hibernate libraries).

The main file is the file `src/demo/TestDrive.gsp`. This is the main entry point to the example. The file extension is `.gsp`, which means this is a Gosu program. A Gosu program is a way to write small scripts or to launch larger Gosu applications that use Gosu classes or other Java types.

The TestDrive.gsp file has a utility function called `configureHibernate`, which sets up a map of configuration properties for Hibernate. It uses the initialization Gosu syntax as follows:

```
var props = {
    "hibernate.dialect" -> "org.hibernate.dialect.HSQLDialect",
    "hibernate.connection.driver_class" -> "org.hsqldb.jdbcDriver",
    "hibernate.connection.url" -> "jdbc:hsqldb:mem:demodb",
    "hibernate.connection.username" -> "sa",
    "hibernate.connection.password" -> "",
    "hibernate.connection.pool_size" -> "1",
    "hibernate.connection.autocommit" -> "true",
    "hibernate.cache.provider_class" -> "org.hibernate.cache.NoCacheProvider",
    "hibernate.hbm2ddl.auto" -> "create-drop",
    //"hibernate.show_sql" -> "true",
    "hibernate.transaction.factory_class" -> "org.hibernate.transaction.JDBCTransactionFactory",
    "hibernate.current_session_context_class" -> "thread"
  }
```

The resulting type is a `java.util.Map<String, String>` object, which is not explicit. Gosu infers the type because of the use of the -> syntax that maps the property (a `String`) to a value (a `String`).

The configuration function then sends these properties to the Hibernate system:

```
var config = new AnnotationConfiguration()
props.eachKeyAndValue( \k, v -> config.setProperty(k, v) )
config.addAnnotatedClass( Book.Type.BackingClass )
return config.buildSessionFactory()
```

Next, the example begins a database transaction using the Hibernate libraries:

```
// Add some books
var sn = sessionFactory.CurrentSession
var tx = sn.beginTransaction()
```

Next, the example creates some new objects to save in the database, and commits them to the database.

```
sn.save( new Book() {:Author="I.N. Herstein", :Title="Abstract Algebra"} )
sn.save( new Book() {:Author="David Hackett Fischer", :Title="Washington's Crossing"} )
tx.commit()
```

Finally, the example runs a database query to confirm that the sample worked successfully:

```
// Find some books
sn = sessionFactory.CurrentSession
tx = sn.beginTransaction()
var query = sn.createQuery( "from " + Book.Type.RelativeName )
for( book in query.iterate() as Iterator<Book> ) {
  print( book.Title + " by " + book.Author )
}
tx.commit()
```

The Gosu code iterates across the results in a `for` loop. Note the code that casts the query to an iterator:

```
as Iterator<Book>
```

By doing this, the code inside the loop can treat the book variable as the *correct specific type*. That means that it can access type-specific fields like `book.Title` and `book.Author` in a type-safe way.

As you can see, inside the loop is an example of the one global function in Gosu, which is the `print` function. It prints a `String` to the console or standard output.

Another project file is `src/demo/Book.gs`. It is a Gosu class that defines the structure of the book objects in the example. Refer to it for details if you plan to implement a Hibernate project in Gosu.

Dynamic Type Example

The Gosu type system has an extensible type system. The language supports custom type loaders that dynamically inject types into the language so you can use them as native objects in Gosu. For example, custom type loaders add Gosu types for objects from XML schemas (XSDs) and from remote WS-I compliant web services (SOAP). Modules of code can create entire namespaces of new types. This means that a type loader can dynamically add types and Gosu code can manipulate them as native objects.

Later versions of the Gosu community release will include more APIs and documentation about creating your own custom type loaders.

The Gosu examples directory includes two example projects that are related:

- `DynamicType`
- `UseDynamicType`

The `DynamicType` project is a demonstration type loader that demonstrates the unusual power of the Gosu extensible type system without having to type a lot of code.

The purpose of `DynamicType` is to create a type with the fully-qualified name `dynamic.Dynamic` that has a few special behaviors in the Gosu type system:

- Any object can be assigned to a variable of this special type. For example, if a variable has type `dynamic.Dynamic`, then you could assign a `String`, a `Map`, an `ArrayList` or any other object to the variable.

Effectively, this is way of creating a non-statically-typed type in Gosu, as opposed to its normal static typing behavior.

- Method invocations and property invocations can happen dynamically and reflectively. In other words, you can call a method on an object or get a property from an object, and at compile time Gosu will permit it.

This dynamic type approach is effectively the opposite of the normal way of coding in a statically-typed language such as Gosu. This is not a normal way of writing Gosu code. This example simply demonstrates that custom type loader behavior can generate complex new and potentially useful behaviors.

The main files of this project are in the directory `src/custom/dyntype`. The Gosu type loaders are written in Java, which is a requirement for Gosu type loaders.

That directory includes the Java files for the type loader for the `Dynamic` type. These Java files tell Gosu how to, for example, load a type, how to get properties from a type, and how to expose and invoke methods on the type. The files have names such as `DynamicMethodInfo.java` and `DynamicTypeLoader.java`.

The `UseDynamicType` example shows how you might use this type in a simple way:

```
classpath ".."
//typeloader custom.dyntype.DynamicTypeLoader

uses dynamic.Dynamic

foo( "hello" )

function foo( value: Dynamic )
{
  print( value.charAt( 1 ) )
}
```

In this example, a function has an argument of the special type called `Dynamic`. The call to the function passes a String object. This compiles successfully because the dynamic type allows any object to be assigned to it. As a function parameter, within the function the parameter name value contains the object. The code then calls the `charAt` method of the object. Normally, Gosu requires that the declared type contain explicit methods on the object if you want to call them. In this case, the example type loader allows any method name on the `Dynamic` object without compile errors. At run time, the type loader handles the method invocation and if the target object at run time contains that method, it will succeed.

In this example, the `charAt` method is a method on a `String` object, so the code succeeds. If you changed the code to call some arbitrary method name that String does not possess, such as `notRealMethodName`, this code would throw an exception at run time. However, because of the special dynamic nature of the type loader, by design it would not show any error at compile time.

WARNING This example is intended for reading only. It is not fully functional in the initial Gosu community release due to in-progress changes in adding type loaders. Later versions of the Gosu community release will include more APIs and documentation about creating your own custom type loaders.

In this example, a function has an argument of the special type called `Dynamic`. The call to the function passes a String object. This compiles successfully because the dynamic type allows any object to be assigned to it. As a function parameter, within the function the parameter name value contains the object. The code then calls the `charAt` method of the object. Normally, Gosu requires that the declared type contain explicit methods on the object if you want to call them. In this case, the example type loader allows any method name on the `Dynamic` object without compile errors. At run time, the type loader handles the method invocation and if the target object at run time contains that method, it will succeed.

In this example, the `charAt` method is a method on a `String` object, so the code succeeds. If you changed the code to call some arbitrary method name that String does not possess, such as `notRealMethodName`, this code would throw an exception at run time. However, because of the special dynamic nature of the type loader, by design it would not show any error at compile time.

WARNING This example is intended for reading only. It is not fully functional in the initial Gosu community release due to in-progress changes in adding type loaders. Later versions of the Gosu community release will include more APIs and documentation about creating your own custom type loaders.

chapter 3

Gosu Programs and Command Line Tools

A *Gosu program* is a file with a `.gsp` file extension that you can run directly from a command-line tool. Additionally, you can run a Gosu program as the main file for a project in an IDE such as IntelliJ IDEA IDE. You can run self-contained Gosu programs using the Gosu command line tool. The Gosu shell command-line tool encapsulates the Gosu language engine. You can run Gosu programs directly from the Windows command line as an interactive session or run Gosu program files. You can optionally edit Gosu code with an included lightweight Gosu editor implemented in Java. Using the graphic Gosu editor is a fast way to use the intelligent Gosu code editor and run code without running a full IDE.

Gosu Command Line Tool Basics

You can use the Gosu shell tool to perform the following tasks:

- Invoke Gosu programs (`.gsp` files). These programs can use other Gosu classes, Gosu extensions, and Java classes.
- Evaluate Gosu expressions interactively using a command-line interface
- Evaluate Gosu expressions passed on the command line

The Gosu shell includes its own lightweight Gosu code editor with intelligent code completion, access to the Gosu type system, and compile time code checking. This version of the code editor simply requires Java, but does not require any other IDEs.

You might consider changing your system's path to add the Gosu shell `bin` directory so you can simply type `gosu` at the command line. On Windows, modify the systemwide `Path` variable by going to the Start menu and choosing Control Panel → System → Environment Variables, and choose the `Path` variable. Add a semicolon and the full path to the `bin` directory in the `gosu` shell directory.

For example, suppose you installed (copied) the Gosu shell directory to the path:

```
C:\gosu\
```

Add the following to the system path:

```
;C:\gosu\bin
```

To test this, close any existing command prompt windows, then open a new command prompt window. Type the following command:

```
gosu -help
```

If the help page appears, the Gosu shell is installed correctly.

Command Line Tool Options

The following table lists the tool command line options:

<table>
<tr><th>Task</th><th>Options</th><th>Example</th></tr>
<tr><td>Run a Gosu program. Include the <code>.gsp</code> file extension when specifying the file name.</td><td><code>filename</code></td><td><code>gosu myfile</code></td></tr>
<tr><td>Open the graphic Gosu editor, which can also run Gosu code. To open a file, include a filename including the <code>.gsp</code> suffix. If you include no filename, the editor creates a new blank unsaved program. Click Save As to save the file.

IMPORTANT: Using the graphic Gosu editor is a fast way to use the intelligent Gosu code editor and run code without running a full IDE.</td><td><code>-g filename</code></td><td><code>-g myfile</code></td></tr>
<tr><td>Evaluate a Gosu expression on command line. Surround the entire expression with quote signs. For any quote sign in the expression, replace it with three double-quote signs. For other special DOS characters such as > and <, precede them with a caret (^) symbol.</td><td>-e <code>expression</code>
-exec <code>expression</code></td><td><code>gosu -e "new DateTime()"</code>
<code>gosu -e """"a"""+"""b""""</code></td></tr>
<tr><td>Add additional paths to the search path for Java classes or Gosu classes. Separate paths with semicolons. If you are running a <code>.gsp</code> file, it is often easier to instead use the <code>classpath</code> command within the <code>.gsp</code> file rather than this option. For related information, see class loading information in “Setting the Class Path to Call Other Gosu or Java Classes” on page 60 and “Advanced Class Loading Registry” on page 62.</td><td><code>-classpath path</code></td><td><code>-classpath C:\gosu\projects\libs</code></td></tr>
<tr><td>Print help for this tool.</td><td><code>-h</code>
<code>-help</code></td><td><code>-h</code></td></tr>
<tr><td>Enter interactive shell. Each line you type runs as a Gosu statement. Any results print to the standard output. To exit, type the <code>exit</code> or <code>quit</code> command. For details, see “Gosu Interactive Shell” on page 63. Also see the standard input option (just a hyphen), discussed later in this table.</td><td><code>-i</code>
<code>-interactive</code></td><td><code>-i</code></td></tr>
<tr><td>Run a Gosu program entered from the standard input stream. Use this to redirect output of one command as Gosu code into the Gosu shell. For a similar feature, refer to “Gosu Interactive Shell” on page 63</td><td>-</td><td>From DOS command prompt:
<code>echo print(new DateTime()) | gosu -</code></td></tr>
</table>

Writing a Simple Gosu Program

The following instructions describe running a basic Gosu program after you have installed the shell.

To write and a run a simple Gosu program

1. Open a command prompt.
2. To open the editor, type the command:
```
gosu -g
```
3. Type the following line into the editor:
```
print( "Hello World" )
```
4. Click Save As
5. Navigate to the directory you want to store the Gosu program and save the file as `helloworld.gsp`.
6. Quit the editor using the close box or the Quit menu item.
7. In a command prompt window, change the working directory to the directory with the file you created.
8. Type the following command:
```
gosu hello_world.gsp
```
The command line will print:
```
Hello World
```

The Structure of a Gosu Program

A simple Gosu program is one or more lines that contain Gosu statements. There are several important other elements of a Gosu program:

- “Metaline as First Line” on page 59
- “Functions in a Gosu Program” on page 59
- “Setting the Class Path to Call Other Gosu or Java Classes” on page 60

Metaline as First Line

Gosu programs support a single line at the beginning of the program for specifying the executable with which to run a file. This is for compliance with the UNIX standard for shell script files. The metaline is optional. If present must the first line of the program. The meta line looks like the following.

```
#!/usr/bin/env gosu
```

Note that the # in the meta line does not mean that the # symbol can start a line comments later on in Gosu programs. The # character is not a valid line comment start symbol.

Functions in a Gosu Program

Your Gosu program can also define functions in the same file and call them.

For example, the following program creates a simple function and calls it twice:

```
print (sum(10,4,7));
print (sum(222,4,3));


function sum (a: int, b: int, c: int) : int {
  return a + b + c;
 }
```

When run, this program outputs:

```
21
229
```

Setting the Class Path to Call Other Gosu or Java Classes

You can call out to any Java or Gosu class as needed. However, you cannot define Gosu classes directly inside your Gosu program file.

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

- Use the `classpath` argument on the command line tool. See “Command Line Arguments” on page 60.
- Add a `classpath` statement to the top of your Gosu program.

The `classpath` statement in a Gosu program improves upon the Java approach, which is to invoke a full and long `classpath` argument option when running the main class.

To add to the class path for a program from within the program, simply add `classpath` statements before all other statements in the program. If you use a metaline (see “Metaline as First Line” on page 59), `classpath` statements appear after the metaline.

A simple version of the classpath statement is simply a relative path in quote signs:

```
classpath "src"
```

If it does not start with a `"/"` character, Gosu treats it **as a relative path**. The path is relative to the folder in which the current program resides. This is the most common use. Use this feature to neatly encapsulate your program and its supporting classes together in one location.

If the path starts with a `"/"` character, Gosu treats it **as an absolute path**.

You can include multiple paths in the same string literal using a comma character as a separator.

Typically you would place Java classes, Gosu classes, or libraries in subdirectories of your main Gosu program.

For example, suppose you have a Gosu program at this location:

```
C:\gosu\myprograms\test1\test.gsp
```

Copy your class file for the class `mypackage.MyClass` to the location:

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

Copy your library files to locations such as:

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

For this example, you would add classpath values with the following statement:

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

Command Line Arguments

There are two ways you can access command line arguments to programs:

- **Manipulating raw arguments.** You can get the full list of arguments as they appear on the command line. If any option has multiple parts separated by space characters (such as `"-username jsmith"`), each component is a separate raw argument.
- **Advanced argument processing.** You can use parse the command line for options with a hyphen prefix and optional additional values associated with the preceding command line option. For example, `"-username jsmith"` is a single option to set the `username` option to the value `jsmith`.

Raw Argument Processing

To get the full list of command line arguments as a list of `String` values, use the `CommandLineAccess` class. Call its `getRawArgs` method, which returns an array of String values.

```
uses gw.lang.cli.CommandLineAccess
print( "CommandLineArgs: " + CommandLineAccess.getRawArgs() )
```

Advanced Argument Processing

A more advanced way to access command line arguments is to write your own class that populate all your properties from the individual command line options. This approach supports Boolean flags or setting values from the command line.

This approach requires you to define a simple Gosu class upon which you define static properties. Define one static property for each command line option. Static properties are properties stored exactly once on the class itself, rather than on instances of the class.

You can then initialize those properties by passing your custom class to the `CommandLineAccess.initialize(...)` method. The `initialize` method overrides the static property values with values extracted from the command line. After processing, you can use an intuitive Gosu property syntax to get the values from the static properties in your own Gosu class.

First, create a Gosu class that defines your properties. It does not need to extend from any particular class. The following example defines two properties, one `String` property named `Name` and an a `boolean` property called `Hidden`:

```
package test
uses gw.lang.cli.*

class Args {
  // String argument
  static var _name : String as Name

  // boolean argument -- no value to set on the command line
  static var _hidden : boolean as Hidden
}
```

Note that the publicly-exposed property name is the symbol after the “`as`” keyword (in this case `Name` and `Hidden`), not the private static variable itself. These are the names that are the options, although the case can vary, such as: “`-name jsmith`” instead of “`-Name jsmith`”.

Choose a directory to save your command line tool. Create a subdirectory named `src`. Inside that create a subdirectory called `test` (the package name). Save this Gosu class file as the file `Args.gs` in that `src/test` directory.

Next, run the following command

Paste in the following code for your program:

```
classpath "src"
uses gw.lang.cli.*
uses test.*

CommandLineAccess.initialize ( Args)

print("hello " + Args.Name)
print("you are " + (Args.Hidden ? "hidden" : "visible") + "!!!!")
```

Click Save As and save this new command line tool as `myaction.gsp` in the directory two levels up from the `Args.gs` file.

From the command batch window, enter the following command

```
gosu myaction.gsp -name John -hidden false
```

This outputs:

```
hello John
you are visible!!!!
```

One nice benefit of this approach is that these properties are available globally to all Gosu code as static properties. After initialization, all Gosu code can access properties merely by accessing the type (the class), without pass a object instance to contain the properties.

Note that you can access the properties uncapitalized to better fit normal command line conventions.

The `String` property we define requires an argument value to follow the option. This is true of all non-boolean property types. However, the boolean property does not require an argument value, and this type is special for this reason. If a property is defined to have type `boolean` and the option is specified with no following value, Gosu assumes the value `true` by default.

The properties can be any type to work with this approach, not merely `String` and `boolean`. However, there must exist a Gosu coercion of the type from a `String` in order to avoid exceptions at run time. If no coercion exists, a workaround is to add a writable property of type `String`,and add a read-only property that transforms that `String` appropriately. This read-only property allows you to do whatever deserialization logic you would like, all defined in Gosu.

Only properties defined with the modifiers `public`, `static`, `writable` properties on your command line class participate in command line argument initialization.

If a user enters an incorrect option, `CommandLineAccess.initialize()` prints a help message and exits with a -1 return code. If you do not want this exit behavior, there is a secondary (overloaded) version of the `initialize` method that you can use instead. Simply add the value `false` a second parameter to the method to suppress exiting on bad arguments.

Special Annotations for Command Line Options

You can use Gosu annotations from the `gw.lang.cli.*` package on the static properties defined in your command line class. Simply add one of the following annotation lines immediately before the line that defines the property:

Annotation	Description
`@Required()`	This command line tool will not parse unless this property is included
`@DefaultValue( String )`	The default string value of this property.
`@ShortName( String )`	The short name of this option when used with a single-dash argument. For example, a property named `Day` preceded by the annotation `@ShortName( "d" )` allows the option `"-d"` shortcut instead of "`-day`". The short name works with the single-dash argument but not the double-dash variant.

For example:

```
package test
uses gw.lang.cli.*

class Args {
// String argument
@Required()
static var _name : String as Name

// boolean argument -- no value to set on the command line
@ShortName("s")
static var _hidden : boolean as Hidden
}
```

Advanced Class Loading Registry

An alternative to using the `classpath` directive directly in the program is to use a `registry.xml` file in the same directory as your program file. The registry file gives you additional control over your Gosu environment, including the ability to specify additional type loaders. It is also useful when you have a lot of programs that share the same configuration environment.

The structure of the `registry.xml` file is as following:

```
<?xml version="1.0" encoding="UTF-8"?>

<serialization xmlns="http://guidewire.com/xml" xmlns:tns="http://guidewire.com/xml"
xmlns:xsd="http://www.w3.org/1999/XMLSchema" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">

  <common-service-init class="gw.internal.gosu.ShellKernelInit"/>
```

```
    <classpath>
      <entry>../lib</entry>
      <entry>../gsrc</entry>
    </classpath>

  </serialization>
```

To add paths to the class path, add more `<entry>` elements containing class paths.

The Self-Contained Gosu Editor

Gosu ships with a simple editor application for writing small Gosu programs. If you run the `gosu` command line tool without any arguments and your run environment supports Java windows programs, the Gosu editor runs.

You can also launch the editor and preload with a specific program using the `-t` flag:

```
C:\eng\pl\carbon\active\gosu>gosu-dist\bin\gosu -t Foo.gsp
```

Gosu Interactive Shell

Gosu includes an interactive text-based shell mode. Each line you type runs as a Gosu statement, and any results print to the standard output.

To enter the interactive shell, run the Gosu batch file in the bin directory with the `-i` option:

```
gosu -i
```

Or, simply run the tool from the command with no extra options to enter interactive mode:

```
gosu
```

The program will display a prompt that indicates that you are in the interactive shell rather than the command prompt environment that called this tool.

```
gs >
```

You can then enter a series of Gosu expressions and statements, including defining functions and variables.

For example, you can type the following series of lines at the prompt:

```
gs > var s = new java.util.ArrayList() {1,2,3}
gs > s.each( \ o : Object -> print(o))
```

The Gosu shell will output the following:

```
1
2
3
```

Type the command `help` to see all available commands in the interactive shell. Additional commands in the interactive shell include the following:

Command	Description
`quit`	Quit the interactive shell.
`exit`	Quit the interactive shell
`ls`	Show a list of all defined variables
`rm` *`VARNAME`*	remove a variable from interactive shell memory
`clear`	clears (removes) all variables from interactive shell memory

If you enter a line of Gosu that necessarily requires additional lines, Gosu displays a different prompt (“`...`”) for you to type the remaining lines. For example, if you type a statement block with an opening brace but no closing brace, you can enter the remaining lines in the statement block. After you enter the line with the closing brace, the shell returns to its regular prompt.

The shell provides code-completion using the `TAB` key. You must type at least one letter of a symbol, after which you can type TAB and the shell will display various options. Note that package completion is not supported.

For example, type the following lines but do not press enter on the last line yet:

```
gs > var s = new java.util.ArrayList() {1,2,3}
gs > s.e
```

If you press TAB, the shell displays properties and methods that begin with the letter "e" and then redisplays the current line you are typing:

```
each( block( java.lang.Object ):void )      eachWithIndex( block( java.lang.Object, int ):void )
elementAt( int )                            ensureCapacity( int )
equals( java.lang.Object )                  except( java.lang.Iterable<java.lang.Object> )

gs > s.e
```

To exit, type the `exit` or `quit` command.

Notes:

- Functions and blocks are supported in the interactive shell. However, defining new Gosu classes is not supported in the interactive shell.
- The interactive shell is different from the *standard in* option for the tool, which may be appropriate for some purposes. You can define the output of one tool to be in Gosu and then redirect (*pipe*) the contents of that tool into the Gosu shell, using the hyphen option.

Helpful APIs for Command Line Gosu Programs

Read Line

Use the `readLine` API to read a line of input from the console using the given prompt. For example:

```
var res = gw.util.Shell.readLine("Are you sure you want to delete that directory?")
```

Is Windows

Call the `gw.util.Shell.isWindows()` method to determine if the current host system is Windows-based.

chapter 4

Gosu Types

This topic describes the Gosu-supported data types and how to use each one. For more information about manipulating types or examining type information at run time, see "Type System" on page 251.

This topic includes:

Built-in Types

Gosu supports the following native data types:

• `Array`	• `Number`
• `Boolean`	• `Object`
• `DateTime`	• `String`
• `Type`	

Array

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

• `Array[]`	• `Number[]`
• `Boolean[]`	• `Object[]`
• `DateTime[]`	• `String[]`

The following example shows a standard indexed array:

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

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

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

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

For more information creating and using arrays, see "Arrays" on page 72.

For more information about special Gosu APIs related to using lists, maps, and other collections in Gosu, see "Collections" on page 183.

Boolean

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

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

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

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

The following values coerce to `false`:

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

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

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

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

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

IMPORTANT For important information about primitives and comparisons of boxed and unboxed types in Gosu, see "Working with Primitive Types" on page 258.

Example

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

DateTime

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

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

Individual characters in the previous table have the following meaning:

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

Other possible values are:

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

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

Example

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

Gosu DateTime and Java

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

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

Gosu implicitly coerces the Gosu `DateTime` object from `String` in most formats. For example:
```
var date : DateTime = "2007-01-02"
```

Number

The `Number` data type represents all numbers, including integers and floating-point values. Gosu supports the standard Java number types of `double`, `float`, `int`, `long`, and `short`. (See "Primitive Types" on page 71.) Possible values for the `Number` data type are:

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

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

IMPORTANT For more information about primitives and comparisons of boxed and unboxed types in Gosu, see "Working with Primitive Types" on page 258.

Scientific Notation

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

1.23×10^{11}

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

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

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

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

Object

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

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

Examples

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

For more information creating and using objects, see “Object Instantiation and Properties” on page 74.

For more information about the Gosu type system, see “Type System” on page 251.

String

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

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

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

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

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

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

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

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

This code outputs:

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

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

Other Methods on String Objects

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

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

This prints:

```
cat
```

String Utilities

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

In-line String Templates

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

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

If you print this variable, Gosu outputs:

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

For more information, see "Gosu Templates" on page 243.

Escaping Special Characters in Strings

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

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

Sequence	Result
`\\`	Inserts a backslash into the string without forcing an escape sequence.
`\"`	Inserts a double-quotation mark into the string without terminating it. **Note:** This does not work inside embedded code within Gosu templates. In such cases, do not escape the double quote characters. See "Gosu Templates" on page 243.
`\n`	Inserts a new line into the string so that the remainder of the text begins on a new line if printed.
`\t`	Inserts a tab into the string to add horizontal space in the line if printed.

Examples

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

Gosu String Templates

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

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

```
${ EXPRESSION }
```

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

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

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

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

If you print this variable, Gosu outputs:

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

This style is the preferred `String` template style.

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

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

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

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

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

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

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

For much more information about Gosu templates, see “Gosu Templates” on page 243.

Type

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

Examples of types

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

For more information about using `Type` objects to get information about a type, see “Type System” on page 251.

Note the following aspects of types:

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

IMPORTANT For more information about the `Type` class and the `typeof` keyword how to use it, see “Basic Type Checking” on page 252.

Primitive Types

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

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

Primitive	Type	Value
`long`	Long integer	64-bit two’s complement
`short`	Short integer	16-bit two’s complement

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

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

IMPORTANT For important information about primitives and comparisons of boxed and unboxed types in Gosu, see “Working with Primitive Types” on page 258.

Access to Java Types

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

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

Arrays

As described previously (“Gosu Types” on page 65), an array can be either indexed or associative.

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

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

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

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

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

Syntax

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

Examples

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

You can also iterate through the members of an array using the `for()` construction. See “Iteration in For() Statements” on page 103 for details.

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

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

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

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

> **IMPORTANT** Consider using collections instead of arrays to take advantage of some advanced features of Gosu. For more information about special Gosu APIs for lists, maps, and other collections in Gosu, see “Collections” on page 183.

Java-based Lists as Arrays

In Gosu, you can Java language list members using standard array index notation. For much more information about Java lists and other collections in Gosu, see “Collections” on page 183. Also see “For() Statements” on page 103 for examples of lists with array-style access syntax.

Example

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

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

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

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

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

The output for this code is:

```
zero
one
two
threeUPDATED
four
```

In some situations, it may be more appropriate to use *collections* such as `List` or `Map` rather than to use arrays. The `List` and `Map` classes inherit from the Java language and Gosu adds additional enhancement methods. For more information about special Gosu APIs related to using lists, maps, and other collections in Gosu, see “Collections” on page 183.

Array Expansion

Gosu supports an operator that expands arrays and lists: the `*.` operator. For more information, see “List and Array Expansion (*.)” on page 187.

Object Instantiation and Properties

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

Note: See “Built-in Types” on page 65 for a list of valid Gosu types.

Creating New Objects

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

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

See “New Object Expressions” on page 86 for more details.

Assigning Object Properties

Property assignment is similar to variable assignment.

Syntax

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

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

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

causes the following error:

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

Example

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

Accessing Object Properties

Gosu retrieves a property’s value using the period operator. You can chain this expression with additional property accessors. For more details about how Gosu handles null values in the expression to the left of the period, see “Handling Null Values In Expressions” on page 95.

Syntax

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

Examples

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

Accessing Object Methods

An object property can be any valid data type, including an array, a function, or another object. An object function is generally called a *method*. Invoking a method on an object is similar to accessing an object property, with the addition of parenthesis at the end to denote the function. Gosu uses the dot notation to call a method on a object instance.

For more details about how Gosu handles `null` values in the expression to the left of the period, see “Handling Null Values In Expressions” on page 95.

Syntax

```
expression.METHOD_NAME()
```

Example

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

See “Static Method Calls” on page 94 for more details. See also “Using Reflection” on page 256 for regarding using type information to determine methods of an object.

Accessing Object Arrays

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

Syntax

```
OBJ[PROPERTY_NAME]
```

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

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

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

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

Examples

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

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

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

Numeric, Binary, and Hex Literals

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

For example:

- Gosu infers that the following variable has type `BigInteger` because the right side of the assignment uses a numeric literal `"1bi"`. That numeric literal means “1, as a big integer”

```
var aBigInt = 1bi
```

- Gosu infers that the following variables have type `Float` because the right side of the assignment uses a numeric literal with an "f" after the number.

```
var aFloat = 1f
var anotherFloat = 1.0f
```

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

Type	Syntax	
`byte`	'b' or 'B' suffix	`var aByte = 1b`
`short`	's' or 'S' suffix	`var aShort = 1s`
`int`	none	`var anInt = 1`
`long`	'l' or 'L' suffix	`var aLong = 1L`
`float`	'f' or 'F' suffix	`var aFloat = 1f` `var anotherFloat = 1.0f`
`double`	'd' or 'D' suffix	`var aDouble = 1d`
`BigInteger`	'bi' or 'BI' suffix	`var aBigInt = 1bi`
`BigDecimal`	'bd' or 'BD' suffix	`var aBigD = 1bd` `var anotherBigD = 1.0bd`

`int`	'0b' or '0B' prefix	`var maskVal1 = 0b0001  // 0 and 1 only`
		`var maskVal2 = 0b0010`
		`var maskVal3 = 0b0100`

int	'0x' or '0X' prefix	`var aColor = 0xFFFF // 0 through F only`

chapter 5

Gosu Operators and Expressions

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

This topic includes:

Gosu Operators

Gosu uses standard programming operators to perform a wide variety of mathematical, logical, and object manipulation operations. If you are familiar with the C, C++ or Java programming languages, you might find that Gosu operators function similar to those other languages. Gosu evaluates operators within an expression or statement in order of precedence. (For details, see "Operator Precedence" on page 78.)

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

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

Operator Precedence

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

Operator	Description

`.`
`[ ]`
`( )`
`?.`
`?[]`
`?:`
	Property access, array indexing, function calls and expression grouping. The operators with the question marks are the null-safe operators. See "Handling Null Values In Expressions" on page 95.

`new`	Object creation, object reflection
`,`	Array value list, as in {*value1*, *value2*, *value3*} Argument list, as in (*parameter1*, *parameter2*, *parameter3*)
`as` `typeas`	As, typeas
`+` `-`	Unary operands (positive, negative values)

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

`typeis`	Typeis

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

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

`+`
`-`
`?+`
`?-`
	Addition, subtraction, string concatenation. The versions with the question marks are the null-safe versions (see "Null-safe Math Operators" on page 96).

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

`==`
`===`
`!=`
`<>`
	Equality, inequality. For general discussion and also comparison of == and ===, see "Equality Expressions" on page 82.

`&`	bitwise AND

Operator	Description
^	bitwise exclusive OR
|	bitwise inclusive OR
&& and	Logical AND, the two variants are equivalent
|| or	Logical OR, the two variants are equivalent
? :	Conditional (ternary, for example, `3*3 == 9 ? true : false`)
= += -= *= /= %= &= ^= |= <<= >>= >>>=	Assignment operator statements. These are technically Gosu *statements*, not *expressions*. For more information, see "Gosu Variables" on page 98.

Standard Gosu Expressions

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

- Arithmetic Expressions
- Conditional Ternary Expressions
- Equality Expressions
- Unary Expressions
- Evaluation Expressions
- Existence Testing Expressions
- Logical Expressions
- New Object Expressions
- Relational Expressions
- Type Cast Expressions (see "Basic Type Checking" on page 252)
- Type Checking Expressions (see "Basic Type Checking" on page 252)
- Conditional Ternary Expressions

Arithmetic Expressions

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

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

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

Addition and Concatenation Operator (+)

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

- If both operands are numeric, the "+" operator performs addition on numeric types.

- If either operand is a `String`, Gosu converts the non-`String` operand to a `String`. The result is the concatenation of the two strings.

Expression	Result
`3 + 5`	`8`
`8 + 7.583`	`15.583`
`"Auto" + "Policy"`	`"AutoPolicy"`
`10 + "5"`	`"105"`
`"Number " + 1`	`"Number 1"`

For the null-safe version of this operator, see “Null-safe Math Operators” on page 96.

Subtraction Operator (-)

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

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

For the null-safe version of this operator, see “Null-safe Math Operators” on page 96.

Multiplication Operator (*)

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

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

For the null-safe version of this operator, see “Null-safe Math Operators” on page 96.

Division Operator (/)

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

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

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

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

For the null-safe version of this operator, see “Null-safe Math Operators” on page 96.

Arithmetic Modulo Operator (%)

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

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

For the null-safe version of this operator, see “Null-safe Math Operators” on page 96.

Bitwise AND (&)

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

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

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

Bitwise Inclusive OR (|)

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

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

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

Bitwise Exclusive OR (^)

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

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

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

Bitwise Left Shift (<<)

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

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

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

Bitwise Right Shift and Preserve Sign (>>)

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

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

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

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

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

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

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

Equality Expressions

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

- == Operator
- != or <> Operator

== Operator

The `==` operator tests for relational equality. The operands can be of any compatible types. The result is always `Boolean`. For reference types, Gosu language, the == operator automatically calls *`object`*`.equals()` to compare values. To compare whether the two operands are the same in-memory object, use the === operator instead. See “=== Operator Compares Object Equality” on page 83 for details.

Syntax

```
a == b
```

Examples

Expression	Result
`7 == 7`	`true`

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

=== Operator Compares Object Equality

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

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

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

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

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

Examples Comparing == and ===

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

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

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

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

!= or <> Operator

The "!=" or "<>" operator tests for relational inequality. The operands can be of any compatible types. The result is always `Boolean`. See also the examples in "Logical NOT" on page 85 for another use of the `!=` operator.

Syntax

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

Examples

Expression	Result
`7 != 7`	`false`

Expression	Result
`"3" <> 3`	`false`
`3 <> 5`	`true`

Evaluation Expressions

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

Syntax

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

Examples

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

Existence Testing Expressions

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

The main way of using an exists expression is to iterate across an array or a list but does not generate. Consider expressions like this as an alternative to simple looping with the Gosu statements `for()`, `while()`, and `do...while()`. The rest of this section focuses on this type of use.

Syntax

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

The index variable *`identifier`* iterates through all possible array index values. The result is the type `Boolean`. The expression returns `true` to indicate success (such an element exists), or returns `false` if no such desired expression exists.

Logical Expressions

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

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

IMPORTANT For important differences between `Boolean` and `boolean` types and differences in coercion, see “Boolean” on page 66.

Gosu supports the following logical expressions:

- Logical AND

- Logical OR
- Logical NOT

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

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

Logical AND

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

Syntax

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

Examples

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

Logical OR

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

Syntax

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

Examples

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

Logical NOT

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

Syntax

```
not a
!a
```

Examples

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

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

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

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

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

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

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

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

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

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

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

- **Preferred example.** Use the following approach for writing code of this type.

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

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

Typeis Expressions

Gosu uses the operator `typeis` to test type information of an object. For more information, see “Basic Type Checking” on page 252.

New Object Expressions

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

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

Syntax In General Case

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

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

Examples

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

Optionally Omit Type Name with 'new' When Type is Determined From Context

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

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

```
// declare variable explicitly with a type
var s : String

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

The following is a more complex example using 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()    // notice the type name is omitted
t = new()    // notice the type name is omitted

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

// optionally OMIT 'new' keyword and still use the Gosu initialization syntax
t.Student = { :Name = "Bob Smith", :Age = 30 }
```

Omitting the `new` keyword can improve readability of creating XML objects when using XSD-based types. Types imported from XSDs sometimes have complex and hard to read type names.

For more information about object initializer syntax, see "Object Initializer Syntax" on page 88

Object Initializer Syntax

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

A simple version looks like the following:

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

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

```
:PROPERTY_NAME = VALUE
```

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

For example, suppose you have the following code:

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

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

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

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

For example, suppose you have the following code:

```
using xsd.test.*

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

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

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

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

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

return simpleTest.toXML()
```

You can instead naturally express it as:

```
using xsd.test.*

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

return simpleTest.toXML()
```

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

Special Syntax for Initializing Lists, Collections, and Maps

There are specialized initializer syntax and rules for creating new lists, collections, and maps, discussed in more detail in “Basic Lists” on page 183 and “Basic HashMaps” 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 consist of the following types:

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

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

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

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

> Operator

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

Syntax

```
expression1 > expression2
```

Examples

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

>= Operator

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

Syntax

```
expression1 >= expression2
```

Examples

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

< Operator

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

Syntax

```
expression1 < expression2
```

Examples

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

<= Operator

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

Syntax

```
expression1 <= expression2
```

Examples

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

Unary Expressions

Gosu supports the following unary (single operand) expressions:

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

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

Numeric Negation

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

Syntax

```
-value
```

Examples

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

Typeof Expressions

Gosu uses the operator `typeof` to determine meta information about an expression's type. The operand can be any valid data type. The result is the *type* of the expression. For more information, see "Basic Type Checking" on page 252.

Bit-wise NOT

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

```
x = !y
```

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

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

Importing Types and Package Namespaces

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

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

Syntax

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

```
uses type
uses namespace
```

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

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

Example 1

The following code uses a fully-qualified type name:

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

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

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

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

Example 2

The following code uses a fully-qualified type name:

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

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

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

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

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

Packages Always in Scope

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

- `soap.*` - only if SOAP type loader is available
- `xsd.*` - only if XML/XSD type loader is available

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

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

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

Conditional Ternary Expressions

The conditional expression (with “?” and “`:`” operators) uses the `Boolean` value of one expression to decide which of two other expressions to evaluate. This logical operator is syntactically right-associative. This means that it groups right-to-left, so that `a ? b:c ? d:e ? f:g` evaluates the same as `a ? b:( c ? d:( e ? f:g ) )`. The second and third operands following the “?” must be of compatible type. (See also “Logical Expressions” on page 84 for details of Boolean logic evaluation.)

Syntax

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

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

For example, given the following conditional expression:

```
condition ? expr1 : expr2
```

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

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

Examples

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

Type of the Result

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

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

For example, consider the following statement:

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

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

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

> **Note:** Although the compile-time type looks at both types to find a common ancestor in the type hierarchy, a ternary clause does not cause the creation of a Gosu *compound type*. For a more detailed explanation of this special type, see “Compound Types” on page 260

Special Gosu Expressions

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

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

Function Calls

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

Syntax

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

Examples

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

Static Method Calls

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

Syntax

```
<type-expression>.<static-method-name>( <argument list> )
```

Examples

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

For more information about static methods and the `static` operator, see “Modifiers” on page 135

Static Property Paths

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

Syntax

```
<type-expression>.<static-property>
```

Examples

Expression	Result
`Claim.TypeInfo`	`Claim typeInfo`
`java.util.Calendar.FRIDAY`	`Friday value`

For more information about static properties in classes, see "Modifiers" on page 135.

Handling Null Values In Expressions

Null-safe Property Access

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

- The standard period operator "`.`", which can access properties or invoke methods.
- The null-safe period operator "`?.`", which can access properties or invoke methods in a null-safe way.

How the Standard Period Operator Handles Null

The standard "." operator is not null-safe.

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

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

At run time, if `claim` or `claim.AssignedGroup` is `null`, Gosu throws a `NullPointerException` exception.

If the expression contains a **method** call, the rules are similar. For example:

```
var groupType = claim.myAction()
```

At run time, if `claim` is `null`, Gosu throws a `NullPointerException` exception.

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 following example uses the null-safe period operator:

```
var s : String = null;

var result = s?.length
```

Although the variable called `result` has the value `null`, the code does not throw any exception. If you use the regular period operator instead, Gosu throws a null pointer exception.

Null-safe Default Operator

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

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

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

Null-safe Indexing for Arrays, Lists, and Maps

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

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

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

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

Null-safe Math Operators

Gosu provides null-safe versions of common math operators.

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

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

chapter 6

Statements

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

This topic includes:

Gosu Statements

A Gosu expression has a value, while Gosu statements do not. Between those two choices, if it is possible to pass the result as an argument to a function, then it is an expression. If it is not possible, then it is a statement.

For example, the following are all Gosu expressions as each results in a value:

```
5 * 6
typeof 42
exists ( var e in Claim.Exposures where e == null )
```

The following are all Gosu statements:

```
print(x * 3 + 5)
for (i in 10) { ... }
if( a == b ) { ... }
```

Note: Do not confuse *statement lists* with *expressions* or *Gosu blocks*. Blocks are anonymous functions that Gosu can pass as objects, even as objects passed as function arguments. For more information, see "Gosu Blocks" on page 165.

Statement Lists

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

It is the Gosu standard always to omit semicolon characters in Gosu at the end of lines. Code is more readable without optional semicolons. In the more rare cases in which you type multiple statement lists on one line, such as within block definitions, use semicolons to separate statements. For other style guidelines, see “General Coding Guidelines” on page 279.

Syntax

```
{ statement-list }
```

Multi-line Example (No semicolons)

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

  print( y )
}
```

Single-line Example (Semicolons)

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

Gosu Variables

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

- Variable Type Declaration
- Variable Assignment

Variable Type Declaration

If a type is specified for a variable, the variable is considered strongly typed, meaning that a type mismatch error results if an incompatible value is assigned to the variable. Similarly, if a variable is initialized with a value, but no type is specified, the variable is strongly typed to the type of the value. The only way to declare a variable without a strong type is to initialize it with a `null` value without a type specified. Note, however, the variable takes on the type of the first non-`null` value assigned to it.

Syntax

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

Examples

```
var age = 42
var age2 : Number
var age3 : Number = "42"
var c : Claim
...
```

Variable Assignment

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

Syntax

```
variable = expression
```

Examples

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

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

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

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

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

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

/=	Divides the value of the variable by the amount on the right-hand side of the operator.	`i = 10`
		`i /= 3`
		Assigns value 3.
		For the `int` type, there is no fraction. If you used a floating-pointing type, the value would be 3.333333.

%=	Divides the value of the variable by the amount on the right-hand side of the operator, and returns the remainder. Next, Gosu assigns this result to the variable on the left-hand side.	`i = 10`
		`i %= 3`
		Assigns value 1.
		This is 10 - (3.3333 as int)*3

&=	Performs a 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.

Operator	Description	Examples

^=	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.

<<=	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 bit-wise left shift of 001010 one bit to the left. The result is binary 101000, which is decimal 40.

Operator	Description	Examples

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

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

`++` unary operator	Adds one to the current value of a variable. Also known as the increment-by-one operator. The unary `++` and `--` operators must always appear after the variable name
	
	IMPORTANT: See related information in "Compound Assignment Compared to Expressions" on page 101.
		`i = 10`
		`i++`
		Assigns value 11.

`--` unary operator	Subtracts one from the current value of a variable. Also known as the decrement-by-one operator. The unary `++` and `--` operators must always appear after the variable name
	
	IMPORTANT: See related information in "Compound Assignment Compared to Expressions" on page 101.
		`i = 10`
		`i--`
		Assigns value 9.

Compound Assignment Compared to Expressions

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

It is important to note that these operators form *statements*, rather than *expressions*.

This means that the following Gosu is valid

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

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

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

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

IMPORTANT Gosu supports the `++` operator after a variable, such as `i++`. Using it before the variable, such as `++i` is unsupported and generates compiler errors.

Gosu Conditional Execution and Looping

Gosu uses the following constructions to perform program flow:

- If() … Else() Statements
- For() Statements
- While() Statements
- Do…While() Statements
- Switch() Statements

If() … Else() Statements

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

Syntax

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

Example

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

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

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

To improve the readability of your Gosu code, Gosu automatically downcasts after a `typeis` expression if the type is a subtype of the original type. This is particularly valuable for `if` statements and similar Gosu structures. Within the Gosu code bounded by the `if` statement, you do not need to do casting (`as` *`TYPE`* expressions) to that subtype. Because Gosu confirms that the object has the more specific subtype, Gosu implicitly considers that variable's type to be the **subtype**, at least within that block of code. For details, see "Basic Type Checking" on page 252

For() Statements

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

Syntax

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

The scope of the `<identifier>` is limited to the statement block itself. The `<expression>` in the `in` clause must evaluate to one of the following:

- `Array`
- Java `List` (or any Java collection)
- Java `Iterator`
- `String` (as a list of characters)

See also "Java-based Lists as Arrays" on page 73 for details on using lists as arrays and accessing list members using array notation.

> **Note:** Gosu provides backwards compatibility for the use of the `foreach(...)` statement. However, it is best to use the `for(...)` statement instead.

Iteration in For() Statements

There are three ways that you can iterate through the members of the list or array contained in the `for()` statement:

- Automatic Iteration
- Automatic Iteration with Index
- Iterator Method Iteration

Automatic Iteration

Use automatic iteration to iterate automatically through the array or list members. Iteration starts with the initial member and continues sequentially until terminating at the last member. Specify this type of iteration by using the following syntax:

```
for ( member in OBJ )
```

In this case, *`OBJ`* must be a list, an array, an interval, or an integer. If it is an integer, Gosu iterates through the list that many times, and the index variable if defined contains the current zero-based index value.

Examples:

```
for( property in Claim.TypeInfo.Properties )
for( iteration in 0..|100 )
for( iteration in 100 )
```

Automatic Iteration with Index

Use index iteration if you need to determine the exact position of a particular element of an array or list. This technique adds an explicit index to determine the index value or to access members of the array or list in a non-sequential fashion using array notation. Specify this type of iteration by using the following syntax:

```
for ( member in OBJ index loopcount )
```

Example:

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

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

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

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

```
  }

  return highIndex

}

//Result
3
```

Iterator Method Iteration

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

Specify this type of iteration by using the following syntax:

```
for(member in object.iterator() )
```

Example

```
//This example iterates over the color values in a map
var mapColorsByName = new java.util.HashMap()

mapColorsByName.put( new java.awt.Color( 1, 0, 0 ), "red" )
mapColorsByName.put( new java.awt.Color( 0, 1, 0 ), "green" )
mapColorsByName.put( new java.awt.Color( 0, 0, 1 ), "blue" )

for( color in mapColorsByName.values().iterator() ) {
  print( color )
}

//Result
red
green
blue
```

Examples

The following examples illustrate the different methods for iterating through the members of an array or list in a `for()` block.

```
// Example 1: Prints all the letters with the index.
for( var letter in gw.api.util.StringUtil.splitWhitespace( "a b c d e" ) index i ) {
  print( "Letter " + i + ": " + letter )
}

// Example 2: Print a message for the first exposure with 'other coverage'.
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 numbers 0 - 99 using simple iteration.
for( i in 100 ) {
  print( i + " of 100" )
}

// Example 4: Prints all Claim properties using reflection.
for( property in Claim.TypeInfo.Properties ) {
  print( property )
}
```

While() Statements

Gosu evaluates the `while()` expression, and uses the Boolean result (it must evaluate to `true` or `false`) to determine the next course of action:

- If the expression is initially `true`, Gosu executes the statements in the statement block repeatedly until the expression becomes `false`. At this point, Gosu exits the `while` statement and continues statement execution at the next statement after the `while()` statement.
- If the expression is initially `false`, Gosu never executes any of the statements in the statement block, and continues statement execution at the next statement after the `while()` statement.

Syntax

```
while( <expression> ) {
  <statements>
}
```

Example

```
// Print the digits
var i = 0

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

Do...While() Statements

The `do...while()` block is similar to the `while()` block in that it evaluates an expression and uses the `Boolean` result to determine the next course of action. The principal difference, however, is the Gosu tests the expression for validity **after** executing the statement block, instead of prior to executing the statement block. This means that the statements in the statement block executes at least once (initially).

- If the expression is initially `true`, Gosu executes the statements in the statement block repeatedly until the expression becomes `false`. At this point, Gosu exits the `do...while()` block and continues statement execution at the next statement after the `do...while()` statement.
- If the expression is initially `false`, Gosu executes the statements in the statement block once, then evaluates the condition. If nothing in the statement block has changed so that the expression still evaluates to `false`, Gosu continues statement execution at the next statement after the `do...while()` block. If action in the statement block causes the expression to evaluate to `true`, Gosu executes the statement block repeatedly until the expression becomes `false`, as in the previous case.

Syntax

```
do {
  <statements>
} while( <expression> )
```

Example

```
// Print the digits
var i = 0

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

Switch() Statements

Gosu evaluates the `switch()` expression, and uses the result to choose one course of action from a set of multiple choices. Gosu evaluate the expression, then iterates through the case expressions in order until it finds a match.

- If a case value equals the expression, Gosu execute its accompanying statement list. Statement execution continues until Gosu encounters a `break` statement, or the switch statement ends. Gosu continues to the next case (Gosu executes multiple case sections) if you omit the `break` statement.
- If no case value equals the expression, Gosu skips to the default case, if one exists. The default case is a case section with the label `default:` rather than `case` *`VALUE`*`:`. The default case must be the last case in the list of sections.

The `switch()` statement block uses a multi-part construction. The `default` statement is optional. However, in most cases, it is best to implement a default case to handle any unexpected conditions.

Syntax

```
switch( <expression> ) {
  case label1 :
    [statementlist1]
```

```
      [break]
    [ ...
    [ case labelN :
      [statementlistN]
      [break] ] ]
    [ default :
      [statementlistDefault]]
  }
```

Example

```
switch( strDigitName ) {
  case "one":
    strOrdinalName = "first"
    break
  case "two":
    strOrdinalName = "second"
    break
  case "three":
    strOrdinalName = "third"
    break
  case "five":
    strOrdinalName = "fifth"
    break
  case "eight":
    strOrdinalName = "eighth"
    break
  case "nine":
    strOrdinalName = "ninth"
    break
  default:
    strOrdinalName = strDigitName + "th"
}
```

To improve the readability of your Gosu code, Gosu automatically downcasts the object after a `typeis` expression if the type is a subtype of the original type. This is particularly valuable for `if` statements and similar Gosu structures such as `switch`. Within the Gosu code bounded by the `if` or `switch` statement, you do not need to do casting (`as` *`TYPE`* expressions) to that subtype for that case. Because Gosu confirms that the object has the more specific subtype, Gosu implicitly considers that variable's type to be the **subtype** for that block of code. There are several special cases that turn off the downcasting. For details, see "Basic Type Checking" on page 252.

Gosu Functions

Functions encapsulate a series of Gosu statements to perform an action and optionally return a value. Generally speaking, functions exist attached to a type. For example, declaring functions within a class. As in other object-oriented languages, *functions declared on a type* are also called *methods*.

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

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

Gosu does not support functions defined within other functions. However, you can use the Gosu feature called blocks to do something similar. See "What Are Blocks?" on page 165 for more information.

Unlike Java, Gosu does not support variable argument functions (so-called `vararg` functions), meaning that Gosu does not support arguments with "`...`" arguments.

Gosu permits you to specify only type literals for a function's return type. Gosu does not support other expressions that might evaluate (indirectly) to a type.

Gosu requires that you provide the return type in the function definition, unless the return type is `void` (no return value). If the return type void, omit the type and the colon before it. Also, any `return` statement must return a type that matches the declared function return type. A missing return type or a mismatched return value generates a compiler error.

Syntax

```
[modifiers] function IDENTIFIER ( argument-declaration-list ) [:type-literal] {
  function-body
}
```

Examples

```
function square( n : Number ) : Number {
  return n * n
}

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

function fibonacci( n : Number ) : Number {
  if (n == 0) { return 0 }
  else if (n == 1 ) { return 1 }
  else {return fibonacci( n - 1 ) + fibonacci( n - 2 ) }
}

function concat ( str1:String, str2:String ) : String {
  return str1 + str2
}
```

IMPORTANT For more information about modifiers that can appear before the word `function` in class definitions, see "Modifiers" on page 135.

If the return type is not `void`, **all** possible code paths must return a value in a method that declares a return type.

In other words, if any code path contains a `return` statement, Gosu requires a `return` statement for all possible paths through the function. The set of all paths includes all outcomes of conditional execution, such as `if` and `switch` statements.

For example, the following method is invalid:

```
//invalid...
class MyClass {
  function myfunction(myParameter) : boolean {
    if myParameter==1
      return true
    if myParameter==2
      return false
  }
}
```

Gosu generates a "Missing Return Statement" error for this function and you must fix this error. The Gosu compiler sees two separate `if` expressions for a total of four total code paths. Even if you believe the function is always used with *`myParameter`* set to value 1 or 2 but no other value, you must fix the error. To fix the error, rewrite the code so that all code paths contain a `return` statement.

For example, you can fix the earlier example using an `else` clause:

```
class MyClass {
  function myfunction(myParameter) : boolean {
    if myParameter==1
      return true
    else
      return false
  }
}
```

Similarly, if you use a `switch` statement, consider using an `else` section.

This strict requirement for `return` statements mirrors the analogous requirements in the Java language.

Named Functions Arguments and Argument Defaults

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

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

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

For example:

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

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

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

```
class MyClass {
var _names : java.util.ArrayList<String>

construct( strings : java.util.ArrayList<String>) {
  _strings = strings
}

  function printWithPrefix( prefix : String = " ---> ") {
    for( n in _strings ) {
      print( prefix + n ) // used a passed-in argument, or use the default " ---> " if omitted
    }
  }
}
```

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

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

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

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

// Alternatively, specify the parameter to pass and override any default if one exists
c.printWithPrefix(:prefix= " next string is:")
```

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

Calling Conventions

When you call a function with a multiple arguments, you can name some of the arguments and not others. Any non-named arguments that you call must match in left-to-right order any **arguments without defaults**.

Gosu considers any additional passed-in non-named arguments as representing the arguments with defaults, passed in the same order (left-to-right) as they are declared in the function.

Public and Private Functions

A function is public by default, meaning that it can be called from any Gosu code. In contrast, a private function can be called only within the library in which it is defined. For example, suppose you have the following two functions defined in a library:

```
public function funcA() {
  ...
}

private function funcB() {
  ...
}
```

Because `funcA()` is defined as public, it can be called from any other Gosu expression. However, `funcB()` is private, and therefore is not valid anywhere except within the library.

For example, a function in another library could call `funcA()`, but it could not call the private `funcB()`. Because `funcA()` is defined in the same library as `funcB()`, however, `funcA()` can call `funcB()`.

Do **not** make any function public without good reason. Therefore, mark a function as private if it is defined only for use inside the library.

IMPORTANT See “Modifiers” on page 135 for more information on class and function level access modifiers.

chapter 7

Intervals

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

What are Intervals?

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

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

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

This prints the following:

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

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

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

```
    i++
}
```

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

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

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

Omitting an Initial or Ending Value

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

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

To do this in Gosu, type the pipe `"|"` character:

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

Compare the following examples:

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

Reversing Interval Order

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

Compare the following examples:

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

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

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

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

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

```
var interv = 5..0
var leftIterator = interv.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 how you defined it).

Granularity (Step and Unit)

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

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

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

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

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

Writing Your Own Interval Type

You can add custom interval types.

There are two basic types of intervals:

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

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

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

The basic properties of an interval are as follows:

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

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

Custom Iterable Intervals Using Sequenceable Items

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

```
package example.pl.gosu.interval

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

Note: For more information about creating enumerations, see “Enumerations” on page 145.

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

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

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

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

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

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

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

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

For more information about Gosu generics, see “Gosu Generics” on page 173.

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

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

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

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

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

  // required methods in ISequenceable interface...

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

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

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

```
    }
}
```

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

```
print("Red to Blue as a closed interval...")
var colorRange = new gw.lang.reflect.interval.SequenceableInterval(
    ColorSequencable.Red, ColorSequencable.Blue, 1, null, true, true, false )

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


print("Red to Blue as an open interval...")
var colorRangeOpen = new gw.lang.reflect.interval.SequenceableInterval(
    ColorSequencable.Red, ColorSequencable.Blue, 1, null, false, false, false )

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

This prints:

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

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

Custom Iterable Intervals Using Manually-written Iterators

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

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

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

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

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

   For more information about Gosu generics, see “Gosu Generics” on page 173.
3. Implement the interface methods for the `Interval` interface.
4. Implement the interface methods for the `Iterable` interface.

The most complex methods to implement correctly are methods that return iterators. The easiest way to implement these methods is to define iterator classes as *inner classes* to your main class. For more information about inner classes, see “Inner Classes” on page 141.

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

Example: Color Interval Written With Manual Iterators

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

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

```
package example.pl.gosu.interval

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

Note: For more information about creating enumerations, see “Enumerations” on page 145.

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

Next, create a new class that extends the following type

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

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

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

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

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

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

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

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

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

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

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

     construct(startAt : int ) {
       _currentIndex = startAt
     }

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

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

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

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

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

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

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

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

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

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

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

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


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

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

This prints:

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

Custom Non-iterable Interval Types

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

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

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

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

chapter 8

Exception Handling

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

This topic includes:

Try-Catch-Finally Constructions

The `try...catch...finally` blocks provides a way to handle some or all of the possible errors that may occur in a given block of code during runtime. If errors occur that the script does not handle, Gosu simply provides its normal error message, as if there was no error handling.

The `try` block contains code where an error can occur, while the `catch` block contains the code to handle any error that does occur.

- If an error occurs in the `try` block, Gosu passes program control to the `catch` block for processing. The initial value of the error-identifier is the value of the error that occurred in the `try` block.
- If an error is thrown from Java code, the value is the exception or error that was thrown. Otherwise, the value is an exception thrown elsewhere in Gosu code.
- If no error occurs, Gosu does not execute the `catch` block.
- If the error cannot be handled in the `catch` block associated with the `try` block where the error occurred, use the `throw` statement. The `throw` statement rethrows the exception to a higher-level error handler.

After all statements in the `try` block have been executed and any error handling has occurred in the `catch` block, the `finally` block is unconditionally executed.

Gosu executes the code inside the `finally` block, even if a `return` statement occurs inside the `try` or `catch` blocks, or if an error is thrown from a `catch` block. Thus, Gosu guarantees that the `finally` block executes.

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

Syntax

```
try
  <try statements>
[catch( exception )
  <catch statements>]
[finally
  <finally statements>]
```

Example

```
try {
  print( "Outer TRY running..." )
  try {
    print( "Nested TRY running..." )
    throw "an error"
  }
  catch( e ) {
    print( "Nested CATCH caught "+e )
    throw e + " rethrown"
  }
  finally { print( "Nested FINALLY running..." ) }
}
catch( e ) { print( "Outer CATCH caught " + e ) }
finally { print( "Outer FINALLY running" ) }
```

Output

```
Outer TRY running...
Nested TRY running...
Nested CATCH caught an error
Nested FINALLY running...
Outer CATCH caught an error rethrown
Outer FINALLY running
```

Throw Statements

The `throw` statement generates an error condition which you can handle through the use of `try…catch…finally` blocks.

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

Syntax

```
throw <expression>
```

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

Example 1

```
try {

  var x = 0

  try {
    if( x == 0 ) { throw "x equals zero" }
    else { throw "x does not equal zero" }
  } catch( e ) {
    if( e == "x equals zero" ) { return( e + " handled locally." ) }
    else { throw e }
  }
} catch( e ) { return( e + " handled higher up." ) }

//Output
x equals zero handled locally.
```

Example 2

```
try {

  var x = 1

  try {
    if( x == 0 ) { throw "x equals zero" }
    else { throw "x does not equal zero" }
  } catch( e ) {
    if( e == "x equals zero" ) { return( e + " handled locally." ) }
    else { throw e }
  }
} catch( e ) { return( e + " handled higher up." ) }

//Output
x does not equal zero handled higher up.
```

Checked Exceptions in Gosu

Gosu allows you to catch and test for catch *checked exceptions*. Checked exceptions identify **specific** types of problems, typically thrown by Java code at some lower level. At a fundamental level, Gosu does not natively distinguish between checked and unchecked exceptions. Generally speaking, use standard unchecked exceptions for designing new code and APIs. However, the language provides a simplified syntax for catching checked exceptions.

The standard syntax for `catch` is simply: `catch(e)` with no declared type for `e` although the exception has type:

```
com.guidewire.commons.gosu.parser.statements.ThrowStatement.GosuThrowException
```

Gosu natively represents *checked* exceptions as an object hanging off of the caught exception in its `Cause` property containing an object of type `Throwable`. The `Cause` property is `null` if the exception has no checked exception associated with it.

The class `Throwable` is the superclass of all errors and exceptions in the Java language. Use the two subclasses `Error` and `Exception` to indicate exceptional situations occurred. Your code can understand the exception by its subclass of those classes, such as the Java `IOException` or `NoSuchMethodException` subclasses.

For example, the following Gosu code catches any specific `IOException` objects stored in the `Cause` property within the exception:

```
try {
  doSomethingThatMayThrowIOException()
}
Catch( e ) {
  if( e.Cause typeis IOException )
  {
    // Handle the IOException, which is the only type this catches
  }
  else
  {
    // rethrow the exception if it is not the right type
    throw e
  }
}
```

However, Gosu provides a concise syntax that lets you catch only specific checked exceptions in an approach similar to Java’s `try`/`catch` syntax. Simply declare the exception of the type of exception you wish to catch:

```
catch( e : ThrowableSubclassName )
```

For example:

```
try {
  doSomethingThatMayThrowIOException()
}
catch( e : IOException ) {
```

```
    // Handle the IOException
  }
```

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

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

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

Object Lifecycle Management ('using' Clauses)

If you have an object with a lifecycle of a finite extent of code, you can simplify your code with the new `using` statement. The `using` statement is a more compact and less error-prone way of working with resources than using `try`/`catch`/`finally` clauses. The cleanup always occurs without requiring a separate `finally` clause, nor do you need to explicitly check whether resources have `null` values. The `using` statement also simplifies synchronization and locking, discussed more later in this section.

For example, to use an output stream typically code would open the stream, then use it, then close it to dispose of related resources. If something goes wrong while using the output stream, your code must close the output stream and perhaps check whether it successfully opened before closing it. In Gosu (or Java) you can use a `try`/`finally` block like the following to clean up the stream:

```
OutputStream os = SetupMyOutputStream() // insert your code that creates your output stream
try {
  //do something with the output stream
}
  finally {
  os.close();
}
```

You can simplify your code using the Gosu `using` statement as follows:

```
using( var os = SetupMyOutputStream() ) {

     //do something with the output stream

} // Gosu disposes of the stream after it completes or if there is an exception
```

The basic form of a using clause is as follows:

```
using( ASSIGNMENT_OR_LIST_OF_STATEMENTS )
{
  // do something here
}
```

The parentheses after the `using` keyword can contain either a Gosu expression or a comma-delimited list of one or more Gosu statements. Gosu runs any statements (including variable assignment) at run time and uses the result as an object to manage in the `using` clause.

Note: You do not need an additional `return` statement to pass the value to the `using` clause. Also note that the statements must be delimited with commas, not semicolons.

There are several categories of objects that work with the `using` keyword: *disposable* objects, *closeable* objects, and *reentrant* objects. If you try to use an object that does not satisfy the requirements of one of these categories, Gosu displays a compile error. The following subtopics discuss these three types of objects.

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

You can return values from `uses` clauses using the standard `return` statement, discussed further in “Returning Values from ‘using’ Clauses” on page 126.

Disposable Objects

Disposable objects are objects that Gosu can dispose to release all system resources. For Gosu to recognize a valid disposable object, the object must have one of the following attributes:

- The object implements the Gosu interface `IDisposable`. This interface contains only a single method called `dispose`. This method takes no arguments. Always use a type that implements `IDisposable` if possible due to faster run time performance.
- The object has a `dispose` method even if it does not implement the `IDisposable` interface. This approach works but is slower at run time because Gosu must use reflection (examining the type at run time) to find the method.

A type’s `dispose` method must release all the resources that it owns. The `dispose` method must release all resources owned by its base types by calling its parent type’s `dispose` method.

To help ensure that resources clean up appropriately even under error conditions, you must design your `dispose` method such that Gosu can call it multiple times without throwing an exception. In other words, if the stream is already closed, then invoking this method has no effect nor throw an exception.

Closeable Objects and ‘using’ Clauses

Closeable objects include objects such as data streams, reader or writer objects, and data channels. Many of the objects in the package `java.io` are closeable objects. For Gosu to recognize a valid closeable object, the object must have one of the following attributes:

- Implements the Java interface `java.io.ICloseable`, which contains only a single method called `close`. This method takes no arguments. Use a type that implements `ICloseable` if possible due to faster run time performance.
- Has a `close` method even if it does not implement the `ICloseable` interface. This approach works but is slower at run time because Gosu must use reflection (examining the type at run time) to find the method.

A type’s `close` method must release all the resources that it owns. The `close` method must release all resources owned by its base types by calling its parent type’s `close` method.

To help ensure that resources clean up appropriately even under error conditions, you must design your `close` method such that Gosu can call it multiple times without throwing an exception. In other words, if the object is already closed, then invoking this method must have no effect nor throw an exception.

The following example creates a new Java file writer instance (`java.io.FileWriter`) and uses the more verbose `try` and `finally` clauses:

```
var writer = new FileWriter( "c:\\temp\\test1.txt" )
try
{
  writer.write( "I am text within a file." )
}
finally
{
  if( writer != null )
  {
    writer.close()
```

```
    }
  }
```

In contrast, you can write more readable Gosu code using the `using` keyword:

```
using( var writer = new FileWriter( "c:\\temp\\test1.txt" ) )
{
  writer.write( "I am text within a file." )
}
```

You can list multiple

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

JDBC Resources and Using Clauses

The following example shows how to use a using clause with a JDBC (Java Database Connection) object.

```
uses java.sql.*

...


function sampleJdbc( con : Connection )
  {
    using( var stmt = con.createStatement(),
           var rs = stmt.executeQuery( "SELECT a, b FROM TABLE2" ) )
    {
      rs.moveToInsertRow()
      rs.updateString( 1, "AINSWORTH" )
      rs.insertRow()
    }
  }
```

Reentrant Objects and ‘using’ Clauses

Re-entrant objects are objects that help manage safe access to data that is shared by re-entrant or concurrent code execution. For example, if you must store data that is shared by multiple threads, ensure that you protect against concurrent access from multiple threads to prevent data corruption. The most prominent type of shared data is class *static variables*, which are variables that are stored on the Gosu class itself.

For Gosu to recognize a valid reentrant object, the object must have one of the following attributes:

- Implements the `java.util.concurrent.locks.Lock` interface. This includes the Java classes in that pacakge: `ReentrantLock`, `ReadWriteLock`, `Condition`.
- Casted to the Gosu interface `IMonitorLock`. You can cast **any** arbitrary object to `IMonitorLock`. This is useful to cast Java monitor locks to this Gosu interface. For more information about monitor locks, refer to:

  `http://en.wikipedia.org/wiki/Monitor_(synchronization)`
- Implements the Gosu class `gw.lang.IReentrant`. This interface contains two methods with no arguments: `enter` and `exit`. Your code must properly lock or synchronize data access as appropriate during the `enter` method and release any locks in the `exit` method.

For blocks of code using locks (code that implements `java.util.concurrent.locks.Lock`), a `using` clause simplifies your code.

The following code uses the `java.util.concurrent.locks.ReentrantLock` class using a longer (non-recommended) form:

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


function useReentrantLockOld() {
  _lock.lock()
  try {
      // do your main work here
    }
  finally {
```

```
      _lock.unlock()
    }
  }
```

In contrast, you can write more readable Gosu code using the `using` keyword:

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


  function useReentrantLockNew() {
    using( _lock ) {
      // do your main work here
    }
  }
```

Similarly, you can cast any object to a monitor lock by adding “`as IMonitorLock`” after the object. For example, the following method call code uses itself (using the special keyword `this`) as the monitor lock:

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

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

Assigning Variables Inside ‘using’ Expression Declaration

The using clause supports assigning a variable inside the declaration of the `using` clause.

This is useful if the expression that you pass to the `using` expression is both:

- something other than a single variable
- you want to reference it from inside the statement list inside the `using` clause declaration

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

To simplify this kind of code, assign the variable before the expression using the `var` keyword:

```
  using ( var VARIABLE_NAME = EXPRESSION ) {
    // code that references the VARIABLE_NAME variable
  }
```

For example:

```
  using( var out = new FileOutputStream( this, false ) ) {
    out.write( content )
  }
```

Passing Multiple Items to the ‘using’ Statement

You can pass multiple items in the using clause expression. Separate each item by a comma character.

For example,

```
  function useReentrantLockNew() {
    using( _lock1, _lock2, _lock3) {
      // do your main work here
    }
  }
```

You can combine the multiple item feature with the ability to assign variables. For more about assigning variables, see “Assigning Variables Inside ‘using’ Expression Declaration” on page 125 .

For example:

```
  using( var lfc = new FileInputStream(this).Channel,
         var rfc = new FileInputStream(that).Channel ) {

        var lbuff  = ByteBuffer.allocate(bufferSize)
        var rbuff  = ByteBuffer.allocate(bufferSize)

        while (lfc.position() < lfc.size()) {
          lfc.read(lbuff)
          rfc.read(rbuff)
```

```
            if (not Arrays.equals(lbuff.array(), rbuff.array()))
            {
              return true
            }

            lbuff.clear()
            rbuff.clear()
          }
          return false
        }
      }
```

Gosu ensures that all objects are properly cleaned up. In other words, for each object to create or resource to acquire, if it creates or acquires successfully, Gosu releases, closes, or disposes the object. Also note that if one of the resources fails to create, Gosu does not attempt to acquire other resources in later-appearing items in the command-seperated list. Instead, Gosu simply releases the ones that did succeed.

IMPORTANT There is much more information about concurrency in the section “Concurrency” on page 269, including other concurrency APIs.

Returning Values from ‘using’ Clauses

You can return values from within `uses` clauses using the standard `return` statement. If you return a value from within a using clause, Gosu considers the clause complete so it calls your object’s final lifecycle management method to clean up your resources. (Gosu calls the `dispose`, `close`, or `exit` method, depending on the type of object.)

The following Gosu example opens a file using the Java `BufferedReader` class and reads lines from the file until the line matches a regular expression. If code in the `while` loop finds a match, it immediately returns the value and skips the rest of the code within the `using` clause.

```
uses java.io.File
uses java.io.BufferedReader
uses java.io.FileReader

function containsText( file : File, regExp : String ) : boolean {
    using( var reader = new BufferedReader( new FileReader( file ) ) ) {
      var line = reader.readLine()
      while( line != null ) {
        if( line.matches( regExp ) ) {
          return true
        }
        line = reader.readLine() // read the next line
      }
    }
    return false
  }
}
```

chapter 9

Classes

Gosu classes encapsulate data and code for a specific purpose. You can subclass and extend existing classes. You can store and access data and functions (also called methods if part of a class) on an instance of the class or on the class itself.

Gosu classes are the foundation for syntax of syntax for interfaces, enumerations, and enhancements. Some of the information in this topic applies to those features as well. For example, the syntax of variables, methods, and modifiers are the same in interfaces, enumerations, and enhancements.

Related topics:

- “Interfaces” on page 147
- “Enumerations” on page 145
- “Enhancements” on page 161

This topic includes:

What Are Classes?

Gosu classes encapsulate data and code to perform a specific task. Typical use of a Gosu class is to write a Gosu class to encapsulate a set of Gosu functions and a set of properties to store within each class *instance*. A class instance is a new in-memory copy of the object of that class. If some Gosu code creates a new instance of the class, Gosu creates the instance in memory with the type matching the class you instantiated. You can manipulate each object instance by getting or setting properties. You can also trigger the class’s Gosu functions. If functions are defined in a class, the functions are also called *methods*.

You can also extend an existing class, which means to make a subclass of the class with new methods or properties or different behaviors than existing implementations in the superclass.

Gosu classes are analogous to Java classes in that they have a package structure that defines the namespace of that class within a larger set of names. For example, if your company is called Smith Company and you were writing utility classes to manipulate addresses, you might create a new class called `NotifyUtils` in the namespace `smithco.utilities`. The fully-qualified name of the class would be `smithco.utilities.NotifyUtils`.

You can write your own custom classes and call these classes from within Gosu, or call built-in classes. You create and reference Gosu classes by name just as you would in Java. For example, suppose you define a class called `Notification` in package `smithco.utilities` with a method (function) called `getName()`.

You can create an instance of the class and then call a method like this:

```
// create an instance of the class
var myInstance = new smithco.utilities.Notification()

// call methods on the instance
var name = myInstance.getName()
```

If desired, you can also define data and methods that belong to the class itself, rather than an instance of the class. This is useful for instance to define a library of functions of similar purpose. The class encapsulates the functions but you never need to create an instance of the class. You can create static methods on a class independent of whether any code ever creates an instance of the class. You are not forced to choose between the two design styles. For more information, see “Static Modifier” on page 140.

If desired, you can write Gosu classes that extend from Java classes. Your class can include Gosu generics features that reference or extend Java classes or subtypes of Java classes. See “Gosu Generics” on page 173 for more information about generics.

Creating and Instantiating Classes

After creating a new class, add additional class variables, properties, and functions to the class. Within a class, functions are also called *methods*. This is standard object-oriented terminology. This documentation refers to functions as methods in contexts in which the functions are part of classes.

If you create a new class, the editor creates a template for a class upon which you can build. The editor creates the package name, class definition, and class constructor. You can add class variables, properties, and functions to the class. Within a class, functions are also called *methods*.

Add variables to a class with the `var` keyword:

```
var myStringInstanceVariable : String
```

You can optionally initialize the variable:

```
var myStringInstanceVariable = "Butter"
```

Define methods with the keyword `function` followed by the method name and the argument list in parentheses, or an empty argument list if there are no arguments to the method. The parameter list is a list of arguments, separated by commas, and of the format:

```
parameterName : typeName
```

For example, the following is a simple method:

```
function doAction(arg1 : String)
```

A simple Gosu class with one instance variable and one public method looks like the following:

```
class MyClass
{
  var myStringInstanceVariable : String

  public function doAction(arg1 : String)
  {
    print("Someone just called the doAction method with arg " + arg1)
```

```
  }
}
```

Constructors

A Gosu class can have a *constructor*, which is like a special method within the class that Gosu calls after creating an instance of that type. For example, if Gosu uses code like "`new MyClass()`", Gosu calls the `MyClass` class's constructor for initialization or other actions. To create a constructor, name the method simply `construct`. For example:

```
class Tree
{
  construct()
  {
    print("A Tree object was just created!")
  }
}
```

If desired, you can delete the class constructor if you do not need it.

Your class might extend another class. If so, it is typically appropriate for your constructor to call its superclass constructor. To do this, use the super keyword. It must be the first line in the subclass constructor. For example

```
class Tree extends Plant
{
  construct()
  {
    super()
    print("A Tree object was just created!")
  }
}
```

If you call `super()`, Gosu calls the superclass no-argument constructor. If you call `super(`*`parameter_list`*`)`, Gosu calls the superclass constructor that matches the matching parameter list. Note that you can call a superclass constructor with different number of arguments or different types than the current constructor.

Static Methods and Variables

If you want to call the method directly on the class itself rather than an instance of the class, you can do this. This feature is called creating a *static method*. Add the keyword `static` before functions that you declare to make them static methods. For example, instead of writing:

```
public function doAction(arg1 : String)
```

Instead use this:

```
static public function doAction(arg1 : String)
```

For more information, see "Static Modifier" on page 140.

Although Gosu supports public variables for compatibility with other languages, it is best to always use *public properties backed by private variables* instead of using public variables.

In other words, in your new Gosu classes use this style of variable declaration:

```
private var _firstName : String as FirstName
```

Do not do this:

```
public var FirstName : String        // do not do this. Public variables are not standard Gosu style
```

For more information about defining properties, see "Properties" on page 130.

> **IMPORTANT** The standard Gosu style is to use public properties backed by private variables instead of using public variables. Do not use public variables in new Gosu classes. See "Properties" on page 130 for more information.

Creating a New Instance of a Class

Typically you want to create an instance of a class. Each instance (in-memory copy) has its own set of data associated with it. The process of constructing a new in-memory instance is called *instantiating a class*. To instantiate a class, use the `new` operator:

```
var e = new smithco.messaging.QueueUtils()
```

You can also use object initializers allow you to set properties on an object immediately after a `new` expression. Use object initializers for compact and clear object declarations. They are especially useful if combined with data structure syntax and nested objects. A simple version looks like the following:

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

For more information on `new` expressions and object initializers, see “New Object Expressions” on page 86.

> **Note:** You can use Gosu classes without creating a new instance of the class using static methods, static variables, and static properties. For more information, see “Static Modifier” on page 140.

Naming Conventions for Packages and Classes

The package name is the namespace for the class, interface, enhancement, enumeration, or other type. Defining a package prevents ambiguity about what class is accessed.

Package names must consist completely of lowercase characters. To access classes or other types in another package namespace, see “Importing Types and Package Namespaces” on page 92. Class names or other type names must always start with an initial capital letter. However, the names may contain additional capital letters later in the name for clarity.

Use the following standard package naming conventions:

Type of class	Package	Example of fully qualified class name
Classes you define	*customername.subpackage*	`smithco.messaging.QueueUtils`

Properties

Gosu classes can define properties, which appear to other objects like variables on the class in that they can use simple intuitive syntax with the period symbol (`.`) to access a property for setting or getting the property. However, you can implement get and set functionality with Gosu code. Although code that gets or sets properties might simply get or set an instance variable, you can implement properties in other more dynamic ways.

To get and set properties from an object with `Field1` and `Field2` properties, just use the period symbol like getting and setting standard variables:

```
// create a new class instance
var a = new MyClass()

// set a property
a.Field1 = 5

// get a property
print (a.Field2)
```

In its most straightforward form, a class defines properties like functions except with the keywords “`property get`” or “`property set`” before it instead of “`function`”. The `get` property function must take zero parameters and the `set` property function always takes exactly one parameter.

For example, the following code defines a property that supports both set and get functionality:

```
class MyClass {
  property get Field3() : String   {
    return "myFirstClass" // in this simple example, do not really return a saved value
  }
  property set Field3(str : String)   {
    print (str) // print only ---- in this simple example, do not save the value
```

```
    }

  }
```

The `set property` function does not save the value in that simple example. In a more typical case, you probably want to create a class instance variable to store the value in a private variable:

```
class MyClass {
  private var _field4 : String

  property get Field4() : String   {
    return _field4
  }
  property set Field4(str : String)   {
    _field4 = str
  }

}
```

Although the data is stored in private variable `_field4`, code that accesses this data does not access the private instance variable directly. Any code that wants to use it simply uses the period symbol (`.`) with the property name:

```
var f = new MyClass()
f.Field4 = "Yes" // sets to "Yes" by calling the set property function
var g = f.Field4 // calls the get property function
```

For some classes, your property getter and setter methods may do very complex calculations or store the data in some other way than as a class variable. However, it is also common to simply get or set a property with data stored as a common instance variable. Gosu provides a shortcut to implement properties as instance variables using *variable alias* syntax using the `as` keyword followed by the property name to access the property. Use this approach to make simple automatic getter and setter property methods backed by an class instance variable.

For example, the following code is functionally identical to the previous example but is much more concise:

```
class MyClass {
  private var _field4 : String as Field4
}
```

The standard Gosu style is to use public properties backed by private variables instead of using public variables.

In other words, write your Gosu classes to look like:

```
private var _firstName : String as FirstName
```

This declares a private variable called `_firstname`, which Gosu exposes as a public property called `FirstName`.

Do not write your classes to look like:

```
public var FirstName : String
```

IMPORTANT The standard Gosu style is to use public properties backed by private variables instead of using public variables. Do not use public variables in new Gosu classes.

Code defined in that class does not need to access the property name. Classes can access their own private variables. In the previous example, other methods in that class could reference `_field4` or `_firstname` variables rather than relying on the property accessors `Field4` or `FirstName`.

Read Only Properties

The default for properties is read-write, but you can make a property read-only by adding the keyword `readonly` before the property name:

```
class MyClass {
  private var _firstname : String as readonly FirstName
}
```

Properties Act Like Data But They Are Dynamic and Virtual Functions

In contrast to standard instance variables, `get property` and `set property` functions are *virtual*, which means you can override them in subclasses and implement them from interfaces. The following illustrates how you would override a property in a subclass and you can even call the superclass's get or set property function:

```
class MyClass
{
  var _easy : String as Easy
}

class MySubClass extends MyClass
{
  override property get Easy() : String
  {
    return super.Easy + " from MySubClass"
  }
}
```

The overridden `property get` function first calls the implicitly defined get function from the superclass, which gets class variable called `_easy`, then appends a string. This get function does **not** change the value of the class variable `_easy`, but code that accesses the `Easy` property from the subclass gets a different value.

For example, if you write the following code in the Gosu Tester:

```
var f = new MyClass()
var b = new MySubClass()

f.Easy = "MyPropValue"
b.Easy = "MyPropValue"

print(f.Easy)
print(b.Easy)
```

This code prints:

```
MyPropValue
MyPropValue from MySubClass
```

Property Paths are Null Tolerant

In Gosu, a period character gets a property from an object or calls a method.

By default, the period operator is not null-safe. This means that if the value on the left side of the period evaluates to `null` at runtime, Gosu throws a *null pointer exception* (NPE). For example, `obj.PropertyA.PropertyB` throws an exception if `obj` or `obj.PropertyA` are null at run time.

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

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

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

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

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

For example, suppose you had a variable called `house`, which contained a property called `Walls`, and that object had a property called `Windows`. You could get the `Windows` value with the following syntax:

```
house.Walls.Windows
```

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

```
// initialize to null
```

```
var x : ArrayList<Windows> = null

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

The following concise Gosu code is equivalent to the previous example and avoids any null pointer exceptions:

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

Null Safe Method Calls

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

For example:

```
house.myaction()
```

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

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

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

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

Null-Safe Versions of Other Operators

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

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

```
var displayName = Book.Title ?: "(Unknown Title)" // return "(Unknown Title)" if Book.Title is null
```

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

```
var book = bookshelf?[bookNumber] // return null if bookshelf is null
```

- The null-safe math operators (`?+`, `?-`, `?*`, `?/`, and `?%`). For example:

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

See “Handling Null Values In Expressions” on page 95.

Design Code for Null Safety

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

You can also design your code to take advantage of this special language feature. For example, expose data as *properties* in Gosu classes and interfaces rather setter and getter methods. This allows you to use the null-safe property operator (the `?.` operator), which can make your code both powerful and concise.

See Also

- For more examples and discussion, see “Handling Null Values In Expressions” on page 95

IMPORTANT Expose public data as properties rather than as getter functions. This allows you to take advantage of Gosu null-safe property accessor paths. Additionally, note it is standard Gosu practice to separate your implementation from your class’s interaction with other code by using properties rather than public instance variables. Gosu provides a simple shortcut with the `as` keyword to expose an instance variable as a property. See “Properties” on page 130

Design APIs Around Null Safe Property Paths

You may also want to design your Gosu code logic around this feature. For example, Gosu uses the `java.util.String` class as its native text class. This class includes a built-in method to check whether the `String` is empty. The method is called `isEmpty`, and Gosu exposes this as the `Empty` property. This is difficult to use with Gosu property accessor paths. For example, consider the following `if` statement:

```
if (obj.StringProperty.Empty)
```

Because `null` coerces implicitly to Boolean (the type of the `Empty` property), the expression evaluates to `false` in either of the following cases:

- if `obj.StringProperty` is null
- the `String` is non-null but its `Empty` property evaluates to `false`.

In typical code, it is important to distinguish these two very different conditions cases. For example, if you wanted to use the value `obj.StringProperty` only if the value is non-empty, it is insufficient to just check the value `obj.StringProperty.Empty`.

To work around this, Gosu adds an enhancement property to `java.util.String` called `HasContent`. This effectively is the reverse of the logic of the `Empty` property. The `HasContent` property only returns `true` if it has content. As a result, you can use property accessor paths such as the following:

```
if (obj.StringProperty.HasContent)
```

Because `null` coerces implicitly to Boolean (the type of the `Empty` property), the expression evaluates to `false` in either of the following cases:

- if `obj.StringProperty` is null
- the `String` is non-null but the string has no content (its `HasContent` property evaluates to `false`).

These cases are much more similar semantically than for the variant that uses `Empty` `(obj.StringProperty.Empty)`. This means you are more likely to rely on path expressions like this.

Be sure to consider null-safety of property paths as you design your code, particularly with Boolean properties.

IMPORTANT Consider null-safety of property paths as you design your code.

Static Properties

You can use properties directly on the class without creating a new instance of the class. For more information, see “Static Modifier” on page 140.

More Property Examples

The following examples illustrate how to create and use Gosu class properties and get/set methods.

There are two classes, one of which extends the other.

The class `myFirstClass`:

```
package mypackage

class MyFirstClass {

  // Explicit property getter for Fred
  property get Fred() : String   {
    return "myFirstClass"
  }

}
```

The class `mySecondClass`:

```
package mypackage

class MySecondClass extends MyFirstClass {
```

```
    // Exposes a public F0 property on _f0
    private var _f0 : String as F0

    // Exposes a public read-only F1 property on _f1
    private var _f1 : String as readonly F1

    // Simple variable with explicit property get/set methods
    private var _f2 : String

    // Explicit property getter for _f2
    property get F2() : String {
      return _f2
    }

    // Explicit property setter for _f2, visible only to classes in this package
    internal property set F2( value : String ) {
      _f2 = value
    }

    // A simple calculated property (not a simple accessor)
    property get Calculation() : Number {
      return 88
    }

    // Overrides MyFirstClass's Fred property getter
    property get Fred() : String {
      return super.Fred + " suffix"

  }
```

Try the following lines in Gosu Tester to test these classes

First, create an instance of your class:

```
var test = new mypackage.MySecondClass()
```

Assign a property value. This internally calls a hidden method to assign `"hello"` to variable `_f0`:

```
test.F0 = "hello"
```

The following line is invalid since f1 is read-only:

```
// This gives a compile error.
test.F1 = "hello"
```

Get a property value. This indirectly calls the `mySecondClass` property getter function for `F2`:

```
print( test.F2 ) // prints null because it is not set yet
```

The following line is invalid because F2 is not visible outside of the package namespace of `MySecondClass`. F2 is publicly read-only.

```
// This gives a compile error.
test.F2 = "hello"
```

Print the `Calculation` property:

```
print( test.Calculation ) // prints 88
```

The following line is invalid since `Calculation` is read-only (it does not have a setter function):

```
//This gives a compiler error.
test.Calculation = 123
```

Demonstrate that properties can be overridden through inheritance because properties are virtual:

```
print( test.Fred ) // prints "myFirstClass suffix"
```

Modifiers

There are several types of modifiers:

- Access Modifiers
- Override Modifier
- Abstract Modifier
- Final Modifier

- Static Modifier

Access Modifiers

You can use access modifier keywords to set the level of access to a Gosu class, interface, enumeration, or a type member (a function, variable, or property). The access level determines whether other classes can use a particular variable or invoke a particular function.

For example, methods and variables marked `public` are visible from other classes in the package. Additionally, because they are public, functions and variables also are visible to all subclasses of the class and to all classes outside the current package. For example, the following code uses the `public` access modifier on a class variable:

```
package com.mycompany.utils

class Test1 {
  public var Name : String
}
```

In contrast, the `internal` access modifier lets the variable be accessed only in the same package as the class:

```
package com.mycompany.utils

class Test2 {
  internal var Name : String
}
```

For example, another class with fully qualified name `com.mycompany.utils.Test2` could access the `Name` variable because it is in the same package. Another class `com.mycompany.integration.Test3` cannot see the `Test.Name` variable because it is not in the same package.

Similarly, modifiers can apply to an entire type, such as a Gosu class:

```
package com.mycompany.utils

internal class Test {
  var Name : String
}
```

Some modifiers only apply to type members (functions, variables, properties, and inner types) and some modifiers apply to type members and top-level types (outer Gosu classes, interfaces, enumerations).

The following table lists the Gosu access modifiers and each one's applicability and visibility:

Modifier	Description	Applies to top-level types	Applies to type members	Visible in class	Visible in package	Visible in subclass	Visible by all
`public`	Fully accessible. No restrictions.	Yes	Yes	Yes	Yes	Yes	Yes
`protected`	Accessible only by types with same package and subtypes.	--	Yes	Yes	Yes	Yes	--
`internal`	Accessible only in same package	Yes	Yes	Yes	Yes	--	--
`private`	Accessible only by the declaring type, such as the Gosu class or interface that defines it.	--	Yes	Yes	--	--	--

If you do not specify a modifier, Gosu assumes the following default access levels:

Element	Default modifier
Types / Classes	`public`
Variables	`private`
Functions	`public`
Properties	`public`

Coding Style Recommendations for Variables

Always prefix `private` and `protected` class variables with an underscore character (_).

Also, avoid public variables. If you are tempted to use public variables, convert the public variables to properties. This separates the way other code accesses the properties from the implementation (the storage and retrieval of the properties). For more style guidelines, see “Coding Style” on page 279.

Override Modifier

Apply the `override` modifier to a function or property implementation to declare that the subtype overrides the implementation of an inherited function or property with the same signature.

For example, the following line might appear in a subtype overriding a `myFunction` method in its superclass:

```
override function myFunction(myParameter : String )
```

If Gosu detects that you are overriding an inherited function or method with the same name but you omit the `override` keyword, you get a compiler warning. Additionally, the Gosu editor offers to automatically insert the modifier if it seems appropriate.

Abstract Modifier

The `abstract` modifier indicates that a type is intended only to be a base type of other types. Typically an abstract type does not provide implementations (actual code to perform the function) for some or all of its functions and properties. This modifier applies to classes, interfaces, functions, and properties.

For example, the following is a simple abstract class:

```
abstract class Vehicle {
}
```

If a type is specified as abstract, Gosu code cannot construct an instance of it. For example, you cannot use code such as `new MyType()` with an abstract type. However, you can instantiate a subtype of the type if the subtype fully implements all abstract members (functions and properties). A subtype that contains implementations for all abstract members of its supertype is referred to as a *concrete type*.

For example, if class `A` is abstract and defines one method’s parameters and return value but does not provide code for it, that method would be declared `abstract`. Another class `B` could extend `A` and implement that method with real code. The class `A` is the abstract class and the class `B` is a concrete subclass of `A`.

An abstract type may contain implementations for none of its members if desired. This means that you cannot construct an instance of it, although you can define a subtype of it and instantiate that type. For example, suppose you write an abstract Gosu class called `Vehicle` which might contain members but no abstract members, it might look like this:

```
package com.mycompany

abstract class Vehicle {
  var _name : String as Name
}
```

You could not construct an instance of this class, but you could define another class that extends it:

```
package com.mycompany

class Truck extends Vehicle {

  // the subtype can add its own members...
  var _TruckLength : int as TruckLength
}
```

You can now use code such as the following to create an instance of `Truck`:

```
var t = new Truck()
```

Things work differently if the supertype (in this case, `Vehicle`) defines abstract members. If the supertype defines abstract methods or abstract properties, the subtype **must** define an *concrete implementation* of each abstract method or property to instantiate of the subclass. A concrete method implementation must implement actual behavior, not just inherit the method signature. A concrete property implementation must implement actual behavior of getting and setting the property, not just inherit the property's name.

The subtype must implement an abstract function or abstract property with the same name as a supertype. Use the `override` keyword to tell Gosu that the subtype overrides an inherited function or method with the same name. If you omit the `override` keyword, Gosu displays a compiler warning. Additionally, the Gosu editor offers to automatically insert the `override` modifier if it seems appropriate.

For example, suppose you expand the `Vehicle` class with abstract members:

```
package com.mycompany

abstract class Vehicle {

  // an abstract property -- every concrete subtype must implement this!
  abstract property get Plate() : String
  abstract property set Plate(newPlate : String)

  // an abstract function/method -- every concrete subtype must implement this!
  abstract function RegisterWithDMV(registrationURL : String)
}
```

A concrete subtype of this `Vehicle` might look like the following:

```
package com.mycompany

class Truck extends com.mycompany.Vehicle
{
  var _TruckLength : int as TruckLength

  /* create a class instance variable that uses the "as ..." syntax to define a property
   * By doing this, you make a concrete implementation of the abstract property "Plate"
   */
  var _licenseplate : String as Plate

  /* implement the function RegisterWithDMV, which is abstract in your supertype, which
   * means that it doesn't define how to implement the method at all, although it does
   * specify the method signature that you must implement to be allowed to be instantiated with "new"
   */
  override function RegisterWithDMV(registrationURL : String ) {
    // here do whatever needs to be done
    print("Pretending to register " + _licenseplate + " to " + registrationURL)
  }
}
```

You can now construct an instance of the concrete subtype `Truck`, even though you cannot directly construct an instance of the supertype `Vehicle` because it is abstract.

You can test these classes using the following code in the Gosu Tester:

```
var t  = new com.mycompany.Truck()
t.Plate = "ABCDEFG"
print("License plate = " + t.Plate)
t.RegisterWithDMV( "http://dmv.ca.gov/register" )
```

This prints the following:

```
License plate = ABCDEFG
Pretending to register ABCDEFG to http://dmv.ca.gov/register
```

Final Modifier

The `final` modifier applies to types, type members, local variables, and function parameters. It specifies that the value of a property, local variable, or parameter cannot be modified after the **initial** value is assigned. The `final` modifier cannot be combined with the `abstract` modifier on anything. These modifiers are mutually exclusive. The `final` modifier implies that there is a concrete implementation and the `abstract` modifier implies that there is no concrete implementation.

Final Types

If you use the `final` modifier on a type, the type cannot be inherited. For example, if a Gosu class is final, you cannot create any subclass of the final class.

The `final` modifier is implicit with *enumerations*, which are an encapsulated list of enumerated constants, and they are implemented like Gosu classes in most ways. For more information, see “Enumerations” on page 145. This means that no Gosu code can subclass an enumeration.

Final Functions and Properties

If you use the `final` modifier with a function or a property, the `final` modifier prevents a subtype from overriding that item. For example, a subclass of a Gosu class cannot reimplement a method defined by its superclass if that function is `final`.

For example, suppose you define a class with final functions and properties:

```
package com.mycompany

class Auto {

  // a final property -- no subtype can reimplement / override this!
  final property get Plate() : String
  final property set Plate(newPlate : String)

  // a final function/method -- no concrete subtype can reimplement / override this!
  final function RegisterWithDMV(registrationURL : String)
}
```

In many ways, properties are implemented like functions in that they are defined with code and they are virtual. Being virtual means properties can be overridden and can call an inherited `get` or `set` property function in their supertype. For more information about properties and shortcuts to define properties backed by instance variables, see “Properties” on page 130.

Final Local Variables

You can use the `final` modifier with a local variable to initialize the value and prevent it from changing.

For example, the following code is valid:

```
class final1
{
  function PrintGreeting() {
   var f = "frozen"
f = "dynamic"
  }
}
```

However, this code is not valid:

```
class final1
{
  function PrintGreeting() {
   final var f = "frozen"
   f = "dynamic" // compile error because it attempts to change a final variable
  }
}
```

If you define a variable as final, you must initialize it with a value immediately as you declare the variable. You cannot declare the variable as final and initialize it in a later statement.

Final Function Parameters

You can use the `final` modifier with a function parameter to prevent it from changing within the function.

For example, the following code is valid:

```
package example

class FinalTest
{
  function SuffixTest( greeting : String) {
     greeting = greeting + "fly"
     print(greeting)
  }
}
```

You can test it with the code:

```
var f = new example.FinalTest()
var s = "Butter"
f.SuffixTest( s )
```

This prints:

```
Butterfly
```

However, if you add the `final` modifier to the parameter, the code generates a compile error because the function attempts to modify the value of a final parameter:

```
class final1
{
  function SuffixTest( final greeting : String) {
     greeting = greeting + "fly"
     print(greeting)
  }
}
```

Static Modifier

Static Variables

Gosu classes can define a variable stored once *per Gosu class*, rather than once *per instance* of the class. This can be used with variables and properties. If a class variable is static, it is referred to as a *static variable*.

> **WARNING** If you use static variables in a multi-threaded environment, you must take special precautions to prevent simultaneous access from different threads. Use static variables sparingly if ever. If you use static variables, be sure you understand *synchronized thread access* fully. For more information, see "Concurrency" on page 269.

To use a Gosu class variable, remember to set its *access level* such as `internal` or `public` so it is accessible to class that need to use it. For more information access levels, see "Access Modifiers" on page 136.

The `static` modifier cannot be combined with the `abstract` modifier. See "Abstract Modifier" on page 137 for more information.

Static Functions and Properties

The `static` modifier can also be used with functions and properties to indicate that it belongs to the type itself rather than instances of the type.

The following example defines a static property and function:

```
class Greeting {

  private static var _name : String

  static property get Name() : String   {
    return _name
  }

  static property set Name(str : String)   {
    _name = str
  }

  static function PrintGreeting() {
    print("Hello World")
  }
}
```

The `Name` property get and set functions and the `PrintGreeting` method are part of the `Greeting` class itself because they are marked as static.

Consequently, this code in the Gosu Tester accesses properties on the class itself, not an instance of the class:

```
Greeting.Name = "initial value"
print(Greeting.Name)
Greeting.PrintGreeting()
```

Notice that this example never constructs a new instance of the `Greeting` class using the `new` keyword.

Static Inner Types

The `static` modifier can also be used with inner types to indicate that it belongs to the type itself (the class itself) rather than a specific instance of the type.

The following example defines a static *inner class* called `FrenchGreeting` within the `Greeting` class:

```
package example

class Greeting
{
  static class FrenchGreeting {
     static public function sayWhat() : String {
       return "Bonjour"
     }
  }

  static public property get Hello() : String {
    return FrenchGreeting.sayWhat()
  }
}
```

You can test this in the Gosu Tester using the code:

```
print(example.Greeting.Hello)
```

This prints:

```
Boujour
```

For more information about this topic, refer to the next section, “Inner Classes” on page 141.

Inner Classes

You can define inner classes in Gosu, similar to inner classes in Java. They are useful for encapsulating code even further within the same file as related code. Use **named inner classes** if you want to be able to refer to the inner class from multiple related methods or multiple related classes. Use **anonymous inner classes** if you just need a simple subclass that you can define in-line within a class method.

Inner classes optionally can include generics features (see “Gosu Generics” on page 173).

Named Inner Classes

You can define a named class within another Gosu class. Once defined, it can be used within the class within which it is defined, or from classes that derive from it. If using it from the current class,

The following example defines a static *inner class* called `FrenchGreeting` within the `Greeting` class:

```
package example

class Greeting
{
  static class FrenchGreeting {
     static public function sayWhat() : String {
       return "bonjour"
     }
  }

  static public property get Hello() : String {
    return FrenchGreeting.sayWhat()
  }
}
```

You can test this in the Gosu Tester using the code:

```
print(example.Greeting.Hello)
```

This prints:

```
bonjour
```

Notice that this example never constructs a new instance of the `Greeting` class or the `FrenchGreeting` class using the `new` keyword. The inner class in this example has the `static` modifier. For more information the static modifier, see “Static Modifier” on page 140.

Similarly, classes that derive from the outer class can use the inner class `FrenchGreeting`. The following example subclasses the `Greeting` class:

```
package example

class OtherGreeting extends Greeting
{
  public function  greetme () {
    var f = new Greeting.FrenchGreeting()
    print(f.sayWhat())
  }
}
```

You can test this code using the following code in the Gosu Tester:

```
var t = new example.OtherGreeting()
t.greetme()
```

This prints:

```
bonjour
```

Anonymous Inner Classes

You can define anonymous inner classes in Gosu from within a class method, similar to usage in Java. The syntax for creating an anonymous inner class is very different from a named inner class. Anonymous inner classes are similar in many ways to creating instances of a class with the `new` operator. However, you can extend a base class by following the class name with braces and then add additional variables or methods. If you do not have another useful base class, use `Object`.

The following is a class that uses an anonymous inner class:

```
package example

class InnerTest {

  static public function runme() {

    // create instance of an anonymous inner class that derives from Object
    var counter = new Object() {

      // anonymous inner classes can have variables (public, private, and so on)
      private var i = 0

      // anonymous inner classes can have constructors
      construct() {
        print("Value is " + i + " at creation!")
      }

      // anonymous inner classes can have methods
      public function incrementMe () {
        i = i + 1
        print("Value is " + i)
      }
    }

    // "counter" is a variable containing an instance of a
    // class that has no name, but derives from Object and
    // adds a private variable and a method

    counter.incrementMe()
    counter.incrementMe()
    counter.incrementMe()
    counter.incrementMe()
    counter.incrementMe()
  }
}
```

You can use the following code in the Gosu Tester to test this class:

```
example.InnerTest.runme()
```

This prints:

```
Value is 0 at creation!
Value is 1
Value is 2
Value is 3
Value is 4
Value is 5
```

Example: Advanced Anonymous Inner Class

The following example shows how to use an anonymous inner class that derives from a more interesting object than `Object`. In this example, the constructor and another method are inherited by the new inner class.

Suppose you define a base class for your inner class and call it `Vehicle`:

```
package example

class Vehicle
{
  construct()
  {
     print("A vehicle was just constructed!")
  }

  function actionOne(s : String) {
     print("actionOne was called with arg " + s)
  }
}
```

You can create a different class that uses `Vehicle` and defines an anonymous inner class based on `Vehicle`:

```
package example

class FancyVehicle
{

  public function testInner() {

    // Create an inner anonymous class that extends Vehicle
    var test = new Vehicle() {
        public function actionTwo(s : String) {
          print("actionTwo was called with arg " + s)
        }
      }
      test.actionOne( "USA" )
      test.actionTwo( "ABCDEFG" )
    }
}
```

Notice that the inner class that defines the `actionTwo` method uses the `new` operator and not the `class` operator. What it actually does, however, is define a new class with no name and then creates one instance of it.

You can test the `FancyVehicle` class with the following code in Gosu Tester:

```
var g = new example.FancyVehicle()
g.testInner()
```

This prints:

```
A vehicle was just constructed!
actionOne was called with arg USA
actionTwo was called with arg ABCDEFG
```

Gosu Block Shortcut for Anonymous Inner Classes Implementing an Interface

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

The parameters of the block are the same number and type as the parameters to the single method. The return type is the same as the return type of that method. This feature works with any interface, including interfaces defined as inner interfaces within a class.

This Gosu block shortcut is helpful for writing concise code in some situations. For example, APIs that use the Java interface called `Runnable`, which is a simple container for code within a method called `run`.

For example, suppose the PluginCallbackHandler class contains an inner interface called PluginCallbackHandler.Block, which implements a run method, similar to the Runnable interface. This interface has one method. Instead of creating an anonymous class to use the inner interface, use a block that takes no arguments and has no return value.

For example, suppose you are using this PluginCallbackHandler class definition in Java:

```
public interface PluginCallbackHandler {

  // DEFINE AN INNER INTERFACE WITHIN THIS CLASS
  public interface Block {
  public void run() throws Throwable;
  }

// ...

public void execute(Block block) throws Throwable;
}
```

This Gosu code creates the anonymous class explicitly:

```
public function messageReceived(final messageId : int) : void {

   // CREATE AN ANONYMOUS CLASS THAT IMPLEMENTS THE INTERFACE
   var myBlock : PluginCallbackHandler.Block = new PluginCallbackHandler.Block() {

      // implement the run() method in the interface
      public function run() : void { /* your Gosu statements here */ }
    };

    // pass the anonymous inner class with the one method
    _callbackHandler.execute(myBlock);
  }
```

However, you can write it more concisely using Gosu block syntax:

```
  public function messageReceived(messageId : int) {

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

For more information about blocks, see “Gosu Blocks” on page 165.

chapter 10

Enumerations

An enumeration is a list of named constants that are encapsulated into a special type of class. Gosu supports enumerations natively, as well as provides compatibility to use enumerations defined in Java.

This topic includes:

Using Enumerations

An enumeration is a list of named constants that are encapsulated into a special type of class. For example, an application tracking cars might want to store the car manufacturer in a property, but track them as named constants that can be checked at compile-time. Gosu supports enumerations natively and also is compatible with enumerations defined in Java.

To create an enumeration

1. Create a class by that name using the same approach you use to create a class.

```
package example

class FruitType {
  construct() {
  }
}
```

2. Change the keyword `class` to `enum` and remove the constructor. Your enumeration now looks like:

```
package example

enum FruitType {

}
```

3. Add your named constants separated by commas:

```
enum FruitType {
  Apple, Orange, Banana, Kiwi, Passionfruit
}
```

Extracting Information from Enumerations

To use the enumerations, simply reference elements of the enumeration class:

```
uses example.FruitType
var myFruitType = FruitType.Banana
```

To extract the name of the enumeration value as a `String`, get its `Name` property. To extract the index of the enumeration value as an `Integer`, get its `Ordinal` property.

For example:

```
print(myFruitType.Name) // prints "Banana"

print(myFruitType.Code) // prints "Banana"

print(myFruitType.Ordinal) // prints "2"
```

Comparing Enumerations

You can compare two enumerations using the == operator. For example,

```
if (myFruitType == FruitType.Apple)
  print("An apple a day keeps the doctor away.")

if (myFruitType == FruitType.Banana)
  print("Watch out for banana peels.")
```

chapter 11

Interfaces

Gosu can define and implement *interfaces* that define a strict contract of interaction and expectation between two or more software elements. From a syntax perspective, interfaces look like class definitions but merely specify a set of required functions necessary for any class that implements the interface. An interface is conceptually a list of method signatures grouped together. Some other piece of code must implement that set of methods to successfully implement that interface. Gosu classes can implement interfaces defined in either Gosu or Java.

This topic includes:

What is an Interface?

Interfaces are a set of required functions necessary for a specific task. Interfaces define a strict contract of interaction and expectation between two or more software elements, while leaving the implementation details to the code that implements the interface. In many cases, the person who writes the interface is different from the person who writes code to implement the interface.

To take a real-world example of an interface, imagine a car stereo system. The buttons, such as for channel up and channel down, are the interface between you and the complex electrical circuits on the inside of the box. You press buttons to change the channel. However, you probably do not care about the implementation details of how the stereo performs those tasks behind the solid walls of the stereo. If you get a new stereo, it has equivalent buttons and matching behavior. Since you interact only with the buttons and the output audio, if the user interface is appropriate and outputs appropriate sounds, the internal details do not matter to you. You do not care about the details of how the stereo internally handles the button presses for channel up, channel down, and volume up.

If a Gosu class implements this interface, Gosu validates at compile time that all required methods are present and that the implementor class has the required method signatures.

An interface appears like a group of related method signatures with empty bodies grouped together for the purpose of some other piece of code implementing the methods. if a class implements the interface, the class agrees to implement all these methods with the appropriate method signatures. The code implementing the interface agrees that each method appropriately performs the desired task if external code calls those methods.

You can write Gosu classes that implement or extend interfaces defined in Gosu or defined in Java.

Defining and Using an Interface

In some ways, interfaces are similar to Gosu classes.

Then, write the rest of the interface like a Gosu class, except that methods are method signatures only with no method bodies. For example, define a simple interface with the following code:

```
interface Restaurant {
  function retrieveMeals() : String[]
  function retrieveMealDetails(dishname : String) : String
}
```

To implement an interface, create a different Gosu class and add "`implements` *`MyInterfaceName`*" after the class name. For example, if your class is called `MyRestaurant`, go to the line:

```
class MyRestaurant
```

Change that line to:

```
class MyRestaurant implements Restaurant
```

If a class implements more than one interface, separate the interface names by commas:

```
class MyRestaurant implements Restaurant, InitializablePlugin
```

In the example `Restaurant` interface, you can implement the interface with a class such as:

```
class MyRestaurant implements Restaurant

  override function retrieveMeals() {
    return {"chicken", "beef", "fish"}
}
  override function retrieveMealDetails(mainitem : String) : String {
    return "Steaming hot " + dishname + " on rice, with a side of asparagus."
  }
}
```

The Gosu editor reveals compilation errors if your class does not properly implement the plugin interface. You must fix these issues.

A common compilation issue is that a method that interface methods that look like properties must be implemented in Gosu explicitly as a Gosu property. In other words, if the interface contains a method whose name starts with `"get"` or `"is"` and takes no parameters, define the method using the Gosu `property` syntax. In this case, do not use the `function` keyword to define it as a standard class method.

For example, if interface `IMyInterface` declares methods `isVisible()` and `getName()`, your plugin implementation of this interface might look like:

```
class MyClass implements IMyInterface {

  property get Visible() : Boolean {
  ...
  }
  property get Name() : String {
  ...
  }
}
```

For more information about properties, see "Defining and Using Properties with Interfaces" on page 149.

If desired, you can write Gosu interfaces that extend from Java interfaces. You can also have your interface include Gosu generics. Your class can extend from Java classes that support generics. Your class can abstract an interface across a type defined in Java or a subtype of such a type. (For more information about generics, see "Gosu Generics" on page 173.)

Defining and Using Properties with Interfaces

Interfaces created in Gosu can declare properties. This means that you can define explicit `property get` or `property set` accessors in interfaces with the following syntax:

```
property get Description() : String
```

Classes can implement an interface property with the explicit `property get` or `property set` syntax.

For example, if the interface is defined as:

```
package example

interface MyInterface
{
  property get VolumeLevel() : int
  property set VolumeLevel(vol : int) : void
}
```

A class could implement this interface with this code:

```
class MyStereo implements MyInterface
{
   var _volume : int

   property set VolumeLevel(vol : int) {
     _volume = vol
   }

   property get VolumeLevel() : int {
    return _volume
  }
}
```

You can test this code in the Gosu tester:

```
uses example.MyStereo

var v = new MyStereo()
v.VolumeLevel = 11
print("the volume goes to " + v.VolumeLevel)
```

If you run this code, it prints:

```
the volume goes to 11
```

Alternatively, a class implementing a property can implement the property using the *variable alias* syntax using the `as` keyword. This language feature lets you make simple get and set methods that use an class instance variable to store the value, and to get the value if anyone requests it.

For example, the following code is functionally identical to the previous example implementation of `MyStereo`, but it is much more concise:

```
uses example.MyStereo
class MyStereo implements MyInterface
{
   var _volume : int as VolumeLevel
}
```

If you run the Gosu tester code as before, it prints the same results.

For information about Gosu class properties in general, see “Classes” on page 127.

Interface Methods that Look Like Properties

If an interface’s methods look like properties, a class implementing an interface must implement the interface in Gosu as a Gosu property using with `property get` or `property set` syntax. In other words, if the interface contains a method whose name starts with "`get`" or "`is`" and takes no parameters, define the method using the Gosu `property` syntax. See earlier in this section for examples.

Modifiers and Interfaces

In many ways, interfaces are defined like classes. One way in which they are similar is the support for modifier keywords. For more information on modifiers, see “Modifiers” on page 135.

One notable differences for interfaces is that the `abstract` modifier is implicit for the interface itself and all methods defined on the interface. Consequently, you cannot use the `final` modifier on the interface or its members.

Superclass Properties

When implementing an interface and referencing a superclasses’ property, use the `super.`*`PropertyName`* syntax, such as:

```
  property get Bar() : String {
... _mySpecialPrivateVar = super.Foo + super.Bar
  }
```

chapter 12

Composition

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

This topic makes extensive references to the following topics:

- "Interfaces" on page 147
- "Classes" on page 127

This topic includes:

Using Gosu Composition

The language feature *composition* allows a class to delegate responsibility for implementing an interface to a different object. This feature helps reuse code easily for some types of projects with complex requirements for shared code. With composition, you do not rely on class inheritance hierarchies to choose where to implement reusable shared code.

Class inheritance is useful for some types of programming problems. However, it can make complex code dependencies fragile. Class inheritance tightly couples a base class and all subclasses. This means that changes to a base class can easily break all subclasses classes. Languages that support multiple inheritance (allowing a type to extend from multiple supertypes) can increase such fragility. For this reason, Gosu does not support multiple inheritance.

What if you have shared behavior that applies to multiple unrelated classes? Since they are unrelated, class inheritance does not naturally apply. Classes with a shared behavior or capability might **not** share a common type inheritance ancestor other than `Object`. Because of this, there is no natural place to implement code that applies to both classes.

Let us consider a general example to illustrate this situation. Suppose you have a window class and a clipboard-support class. Suppose you have a user interface system with different types of objects and capabilities. However, some of the capabilities might not correspond directly to the class inheritance. For example, suppose you have classes for visual items like windows and buttons and scroll bars. However, only some of these items might interact with the clipboard copy and paste commands.

If not all user interface items do not support the clipboard, you might not want to implement your clipboard-supporting code in the root class for your user interface items. However, where do you put the clipboard-related code if you want to write a window-handling class that is also a clipboard part? One way to do this is to define a new interface that describes what methods each class must implement to support clipboard behavior. Each class that uses this interface implements the interface with behavior uniquely appropriate to each class. This is an example of sharing a behavioral contract defined by the interface. However, each implementation is different within each class implementation.

What if the actual implementation code for the clipboard part is identical for each class that uses this shared behavior? Ideally, you write shared code only **once** so you have maximum encapsulation and minimal duplication of code. In some cases there does not exist a shared root class other than `Object`, so it might not be an option to put the code there. If Gosu supported multiple inheritance, you could encapsulate the shared code in its own class and classes could inherit from that class in addition to any other supertype.

Fortunately, you can get many of the benefits of multiple inheritance using another design pattern called *composition*. Composition encapsulates implementation code for shared behavior such that calling a method on the main object forwards method invocations to a subobject to handle the methods required by the interface.

Let us use our previous example with clipboard parts and windows. Let us suppose you want to create a subclass of window but that implements the behaviors associated with a clipboard part. First, create an interface that describes the required methods that you expect a clipboard-supporting object to support, and call it `IClipboardPart`. Next, create an implementation class that implements that interface, and call it `ClipboardPart`. Next, create a window subclass that implements the interface and delegates the actual work to a `ClipboardPart` instance associated with your window subclass.

The delegation step requires the Gosu keyword `delegate` within your class variable definitions. Declaring a delegate is like declaring a special type of class variable.

The `delegate` keyword has the following syntax:

```
delegate PRIVATE_VARIABLE_NAME represents INTERFACE_LIST
```

Or optionally

```
delegate PRIVATE_VARIABLE_NAME : TYPE represents INTERFACE_LIST
```

The *`INTERFACE_LIST`* is a list of one or more interface names, with commas separating multiple interfaces.

For example:

```
delegate _clipboardPart represents IClipboardPart
```

Within the class constructor, create an instance of an object that implements the interface. For example:

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

After that point in time, Gosu intercepts any method invocations on the object for that interface and forward the method invocation to the delegated object.

Let us look at complete code for this example.

The interface:

```
package test

interface IClipboardPart
{
  function canCopy() : boolean
  function copy() : void
  function canPaste() : boolean
  function paste() : void
```

```
}
```

The delegate implementation class:

```
package test

class ClipboardPart implements IClipboardPart {
  var _myOwner : Object

  construct(owner : Object) {
    _myOwner = owner
  }

  // this is an ACTUAL implementation of these methods...
  override function canCopy() : boolean { return true }
  override function copy() : void { print("Copied!")}
  override function canPaste() : boolean { return true }
  override function paste() : void { print("Pasted!") }

}
```

Your class that delegates the `IClipboardPart` implementation to another class

```
package test

class MyWindow implements IClipboardPart {
  delegate _clipboardPart represents IClipboardPart

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

Finally, enter the following code into the Gosu Tester:

```
uses test.MyWindow

var a = new MyWindow()

// call a method handled on the delegate
a.paste()
```

It prints:

```
Pasted!
```

Overriding Methods Independent of the Delegate Class

You can override any of the interface methods that you delegated. Using the previous example, if the `canCopy` method is in the delegate interface, your `MyWindow` class can choose to override the `canCopy` method to specially handle it. For example, you could trigger different code or choose whether to delegate that method call.

For example, your `MyWindow` class can override a method implementation using the override keyword, and calling the private variable for your delegate if desired:

```
override function canCopy() : boolean
{
  return someCondition && _clipboardPart.canCopy();
}
```

Declaring Delegate Implementation Type in the Variable Definition

You can declare a delegate with an explicit type for the implementation class. This is particularly valuable if any of your code accessing the delegate directly in terms of the implementation class. For example, by declaring the type explicitly, you can avoid casting before calling methods on the implementation class that you know are not defined in the interface it implements.

To declare the type directly, add the implementation type name followed by the keyword `represents` before the interface name. In other words, use the following syntax:

```
private delegate PRIVATE_VARIABLE_NAME : IMPLEMENTATION_CLASS represents INTERFACE_NAME
```

For example,

```
private delegate _clipboardPart : ClipboardPart represents IClipboardPart
```

Using One Delegate for Multiple Interfaces

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

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

You might notice that in this example the line does not specify an explicit type for `_employee` and yet it represents **two** different types (in this case, two interface types). You might wonder about the compile-time type of the variable called `_employee`. Because the variable must satisfy all requirements of both types, Gosu uses a special type called a *compound type*. A literal of this type is expressed in Gosu as a list separated by the ampersand symbol (&). For example:

```
ISalariedEmployee & IOfficer
```

Typical code does not need to mention a compound type explicitly. However, remember this syntax in case you see it during debugging code that uses the `delegate` keyword with multiple interfaces.

For more details of compound types, see "Compound Types" on page 260.

Using Composition With Built-in Interfaces

You can use composition with any interfaces, including built-in interfaces. For example, you could give a custom object all the methods of `java.util.List` and delegate the implementation to an instance of `java.util.ArrayList` or another `List` implementation.

For example:

```
class MyStringList implements List<String>
{
  delegate _internalList represents List<String> = new ArrayList<String>()
}
```

You could now use this class and call any method defined on the `List` interface:

```
var x = new MyStringList()
x.add( "TestString" )
```

chapter 13

Annotations

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

This topic includes:

Annotating a Class, Method, Type, or Constructor

Annotations are a simple syntax to add metadata to a Gosu class, constructor, method, or property. For example, annotations could add indicate what a method returns, or indicate what kinds of exceptions the method might throw. You can add completely custom annotations and this information can be read at run time. If you use an annotation, use the at sign (`@`), followed by the annotation name, immediately before declarations of what they annotate.

For example, the following simple example specifies a class to expose as a web service for external systems:

```
@WsiWebService
class MyServiceAPI {
  public function myRemoteMethod() {}
}
```

In some cases, you follow the annotation name with an argument list within parentheses. The following example specifies a function might throw a specific exception using arguments to the annotation:

```
class MyClass{

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

The annotation may not require any arguments, or the arguments may be optional. If so, you can omit the parentheses. For example, suppose you add an annotation called `MyAnnotation` that takes no arguments. You could use it in the following (verbose) syntax:

```
@MyAnnotation()
```

Since there are no arguments, you can optionally omit the parentheses:

```
@MyAnnotation
```

You can use annotations defined natively in Gosu or directly use Java annotations.

Argument List Notes

Gosu requires argument lists to be in the same format as regular function or method argument lists:

```
// standard Gosu argument lists
@KnownBreak("user", "branch", "ABC-xxxxx")
```

Gosu supports the named arguments calling convention from Java:

```
@KnownBreak(targetUser = "user", targetBranch = "branch", jira = "ABC-xxxxx")
```

For related information about named arguments, see “Named Functions Arguments and Argument Defaults” on page 108.

Built-in Annotations

The Gosu language includes built-in annotations defined in the `gw.lang.*` package, which is always in scope, so their fully-qualified name is not required.

The following table lists the built-in general annotations:

Annotation	Description	Usage limits	Parameters
`@Param`	Specifies the documentation of a parameter.	Methods only	(1) The name of the parameter. (2) Documentation in Javadoc format for the method’s parameter.
`@Returns`	Specifies the documentation for the return result of the method.	Methods only, but only once per method	(1) Documentation in Javadoc format for the method’s return value.
`@Throws`	Specifies what exceptions might be thrown by this method.	Methods only	(1) An exception type. (2) A description in Javadoc format of what circumstances it would throw that exception, and how to interpret that exception.
`@Deprecated`	Specifies not to use a class, method, constructor, or property. It goes away in a future release. Begin rewriting code to avoid using this class, method, constructor, or property.	Can appear anywhere, but only once for any specific class, method, constructor, or property.	(1) A warning string to display if this deprecated class, method, or constructor is used.

The following code defines a class that uses several built-in annotations:

```
package com.mycompany
uses java.lang.Exception

@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*, which means it was a valid API but not anymore. A deprecated API is temporarily available but a future release will remove it. Immediately start to refactor code that uses deprecated APIs. This ensures your code is compatible with future releases, which will simplify your upgrades.

```
class MyClass {

  @Deprecated("Don't use MyClass.myMethod(). Instead, use betterMethod()")
  public function myMethod() {print("Hello")}

  public function betterMethod() {print("Hello, World!")}
}
```

Because annotations are implemented as Gosu classes (see "Defining Your Own Annotations" on page 158), the annotation class that you are implicitly using must be in the current Gosu scope. You can ensure that it is in scope by fully qualifying the annotation. For example, if the `SomeAnnotation` annotation is defined within the package `com.mycompany.some.package`, specify the annotation like:

```
@com.mycompany.some.package.SomeAnnotation
class SomeClass {
  ...
}
```

Alternatively, import the package using the Gosu `uses` statement and then use the annotation more naturally and concisely by using only its name:

```
uses com.mycompany.some.package.SomeAnnotation.*

@SomeAnnotation
class SomeClass {
  ...
}
```

Annotations at Run Time

You can get annotation information from a class either directly by getting the type from an object at runtime. You can get an object's type at runtime using the `typeof` operator, such as: `typeof` *`TYPE`*

You can get annotation information from a type, a constructor, a method, or a property by accessing their type information objects attached to the type. You can call the `getAnnotation` method to get all instances of specific annotation, as a list of annotation instances. In the examples in the table, the variable `i` represents the index in the list. In practice, you would probably search for it by name using `List` methods like `list.firstWhere(\ s -> s.Name = "MethodName")`.

Get annotations on a specific instance of a...	Example using the @Deprecated annotation
Type	`(typeof obj).TypeInfo.getAnnotation(Deprecated)`
Constructor	`(typeof obj).TypeInfo.Constructors[i].getAnnotation(Deprecated)`
Method	`(typeof obj).TypeInfo.Methods[i].getAnnotation(Deprecated)`
Property	`(typeof obj).TypeInfo.Properties[i].getAnnotation(Deprecated)`

Using these methods, the return result is automatically statically typed as a list of the proper type. Using the examples in the previous table, the result would be of type:

```
List<Deprecated>
```

This type is shown using generics syntax, and it means "a list of instances of the `Deprecated` annotation class". For more information about generics, see "Gosu Generics" on page 173.

You can additionally get all annotations (not just one annotation type) using the two properties `Annotations` and `DeclaredAnnotations`. These two properties are slightly different and resemble the Java versions of annotations with the same name. On types and interfaces, `Annotations` returns all annotations on this type/interface and on all its supertypes/superinterfaces. `DeclaredAnnotations` returns annotations only on the given types, ignoring supertypes/superinterfaces. In constructors, properties, and methods, the `Annotations` and `DeclaredAnnotations` properties return the same thing: all annotations including supertypes/superinterfaces. In the examples in the table, the variable `i` represents the index in the list. In practice, you would probably search for it by name using `List` methods like `list.firstWhere(\ s -> s.Name = "MethodName")`.

Get all annotations on...	Example
Type	`(typeof obj).TypeInfo.Annotations`
Constructor	`(typeof obj).TypeInfo.Constructors[i].Annotations`
Method	`(typeof obj).TypeInfo.Methods[i].Annotations`
Property	`(typeof obj).TypeInfo.Properties[i].Annotations`

For a detailed example of accessing annotations at run time, see “Defining Your Own Annotations” on page 158.

Defining Your Own Annotations

You can define new annotations to add entirely new metadata annotations, apply them to various kinds of programming declarations, and then retrieve this information at run time. You can also get information at run time about objects annotated with built-in annotations. For example, you could mark a Gosu class with metadata and retrieve it at run time.

Annotations are implemented as Gosu classes, and an annotation is simply a call to the annotation class’s constructor. A class constructor is similar to a class method. However, Gosu automatically calls the constructor if it creates a new instance of the class, such as if Gosu code uses the `new` keyword.

You can define new annotation types that can be used throughout Gosu. Annotations are defined just like classes except they must extend the interface `IAnnotation`. The `IAnnotation` interface is a marker interface that designates a class as an *annotation definition*.

Suppose you want a new annotation that allows us to annotate which people wrote a Gosu class. You could use the annotation at run time for debugging information or to file a bug in certain error conditions. To do this, you can create an annotation called `Author`.

For example, the following example defines a new annotation `Author` in the `com.mycompany` package

```
package com.mycompany

class Author implements IAnnotation {
}
```

In this case the annotation has no constructor, which implies the annotation takes no parameters, so it could simply be called as:

```
@Author()
```

Because there are no arguments, you can omit the parentheses:

```
@Author
```

However, as written in this example so far, you used the annotation but not specified any authors. Annotations can define arguments so you can pass information to the annotation, which might stored in private variables. Annotations can have properties or arguments of any type. However, if defining properties and arguments, be careful you never define circular references between annotation classes and regular classes.

This example requires only a single `String` argument, so define the annotation `Author` to take one argument to its constructor. Gosu calls the constructor once for the type after initializing Gosu at run time. In your constructor, save the constructor arguments value in a private variable:

```
package com.guidewire.pl.docexamples.annotations
```

```
class Author implements IAnnotation
{
  // Define a public property Author, backed by private var named _author
  private var _author : String as AuthorName

  construct(a : String)
  {
    // The constructor takes a String, which means the Author of this item
    _author = a;
  }
}
```

In this example, the annotation saves the `String` argument in a class instance variable called `_author`. Because of the phrase "`as Author`" in the definition of the variable, at run time you can extract this information as the annotation's public property `Author`.

By default, this annotation can be used on any method, type, property, or constructor, and as many times as desired. For example, you could specify multiple authors for a class or even multiple authors for methods on a class, or both. You can customize these settings and restrict the annotation's usage, as discussed in "Customizing Annotation Usage" on page 160.

For now, test this annotation by using it on a newly defined type, such as a new Gosu class. Create the following class in the `com.mycompany` package:

```
package com.guidewire.pl.docexamples.annotations

uses com.guidewire.pl.docexamples.annotations.Author

@Author("A. C. Clarke")
@Author("J. M. Straczynski")
class Spaceship {

}
```

You can get annotation information from a class either directly by getting the type from an object at runtime. First can get an object's type at runtime using the `typeof` operator or by getting the `Type` property from an object. Next, get its `TypeInfo` property and call the `getAnnotation` method, passing your annotation class name directly as a parameter. The result is a list (`java.util.List`) containing annotation information objects. For each one of these, get its `Value` property and coerce it to your annotation class.

For example, add the example classes from earlier in this topic into your Gosu environment and then paste the following code into the Gosu Tester:

```
uses com.guidewire.pl.docexamples.annotations.Author
uses com.guidewire.pl.docexamples.annotations.Spaceship

var sub = new Spaceship()

print("Get annotations from an object's type, then iterate with 'for'...")
var annotations = (typeof sub).TypeInfo.getAnnotation(Author)
for (a in annotations)  {
  print("  Author: " + (a.Value as Author).AuthorName);
}

print("")
print("Get annotations directly from a type, then iterate with a block...")
var annotations2 = Spaceship.Type.TypeInfo.getAnnotation(Author)
annotations2.each( \ a -> print("  Author: " + (a.Value as Author).AuthorName))
```

This example prints the following:

```
Get annotations from an object's type, then iterate with 'for'...
  Author: A. C. Clarke
  Author: J. M. Straczynski

Get annotations directly from a type, then iterate with blocks...
  Author: A. C. Clarke
  Author: J. M. Straczynski
```

For more information about blocks and using collections, see "Gosu Blocks" on page 165 and "Collections" on page 183.

Customizing Annotation Usage

Usage of each annotation can be customized, such as allowing it under certain conditions. For example, notice that the built-in annotation `@Returns` can appear only on methods. To restrict, usage like this, use the `AnnotationUsage` meta-annotation within your annotation definition. The `AnnotationUsage` meta-annotation takes two parameters, the *target* and the *modifier*.

The target defines where the annotation can be used using these enumerations:

- `annotation.UsageTarget.Method` - This annotation can be used on a method.
- `annotation.UsageTarget.Type` - This annotation can be used on a type, including classes
- `annotation.UsageTarget.Property` - This annotation can be used on a property.
- `annotation.UsageTarget.Constructor` - This annotation can be used on a constructor.

The modifier defines how many times the annotation can be used (for that target) using these enumerations:

- `annotation.UsageModifier.None` - This annotation cannot exist on that target
- `annotation.UsageModifier.Once` - This annotation can only appear once on that target
- `annotation.UsageModifier.Many` - This annotation can appear many (unlimited) times on that target

For example, the `@Returns` annotation can only appear on methods, and can only appear once, so it specifies its requirements with this line right before its annotation definition:

```
@AnnotationUsage(annotation.UsageTarget.Method, annotation.UsageModifier.One)
```

The default availability is universal. In other words, if no `AnnotationUsage` attribute is defined on an annotation, the usage defaults to allow the annotation **unlimited** times on **all** parts of a type or class.

However, once any `AnnotationUsage` annotation is used in an annotation definition, all targets default to `None`. After using `AnnotationUsage` once, Gosu requires you to explicitly specify supported targets using `AnnotationUsage` meta-annotations. You can optionally add multiple lines for each type of permitted use.

> **IMPORTANT** The default annotation availability is universal (all parts, many times). As soon as you use one `AnnotationUsage` line in the annotation definition, Gosu assumes all targets revert to `None.` Explicitly list all permitted usages with `AnnotationUsage` lines as appropriate.

The `annotation` class is always in scope. You do not need to fully-qualify the class name or use a `uses` statement in files that use it.

chapter 14

Enhancements

Gosu enhancements are a language feature that allows you to augment classes and other types with additional concrete methods and properties. For example, use enhancements to define additional utility methods on a class or interface that cannot be directly modified, even code written in Java. You can enhance classes originally written in Gosu or Java. *Enhancing* is different from *subclassing* in important ways. Enhancing a class makes new methods and properties available to **all** objects of that enhanced type, not just Gosu code that explicitly knows about the subclass. Use enhancements to add powerful functionality omitted by the original authors.

This topic includes:

Using Enhancements

Gosu *enhancements* allow you to augment classes and other types with additional concrete methods and properties. The most valuable use of this feature is to define additional utility methods on a Java class or interface that cannot be directly modified. This is most useful if a class's source code is unavailable, or a given class is *final* (cannot be subclassed). Enhancements can be used with interfaces as well as classes, which means you can add useful methods to interfaces.

Enhancing a class or other type is different from subclassing: enhancing a class makes the new methods and properties available to **all** instances of that class, not merely subclass instances. For example, if you add an enhancement method to the `String` class, all `String` instances in Gosu automatically have the additional method.

You can also use enhancements to overcome the language shortcomings of Java or other languages defining a class or interface. For example, Java-based classes and interfaces can be used from Gosu, but they do not natively allow use of blocks, which are anonymous functions defined in-line within another function. (See "Gosu Blocks" on page 165.) Gosu includes many built-in enhancements to commonly-used Java classes in its products so that any Gosu code can use them.

For example, Gosu extends the Java class `java.util.ArrayList` so you can use concise Gosu syntax to sort, find, and map members of a list. These list enhancements add additional methods to Java lists that take Gosu blocks as parameters. The original Java class does not support blocks because the Java language does not support blocks. However, these enhancements add utilities without direct modifications to the class. Gosu makes these additional methods automatically and universally available for all places where Gosu code uses `java.util.ArrayList`.

You can also enhance an interface. This does **not** mean an enhancement can add new methods to the interface itself. The enhancement does not add new requirements for classes to implement the interface. Instead, enhancing an interface means that all objects whose class implements the interface now has new methods and properties. For example, if you enhance the `java.util.Collection` interface with a new method, all collection types suddenly have your newly-added method.

This does not go into detail about the built-in enhancements to collections. For reference documentation, see "Collections" on page 183. If you have not yet learned about Gosu *blocks*, you may want to first review "Gosu Blocks" on page 165.

Syntax for Using Enhancements

There is no special syntax for using an already-defined enhancement. The new methods and properties are automatically available within the Gosu editor for all Gosu contexts.

For example, suppose there is an enhancement on the `String` type for an additional method called `calculateHash`. Use typical method syntax to call the method with any `String` object accessible from Gosu:

```
var s1 = "a string"
var r1 = s1.calculateHash()
```

You could even use the method on a value you provide at compile time:

```
"a string".calculateHash()
```

Similarly, if the enhancement adds a property called `MyProperty` to the `String` type, you could use code such as:

```
var p = "a string".MyProperty
```

The new methods and properties all appear in the list of methods that appears if you type a period (.) character in the Gosu editor. For example, if typing "`s1.calculateHash()`", after you type "`s1.`" the list that appears displays the `calculateHash` method as a choice.

Creating a New Enhancement

To create a new enhancement, put the file in your Gosu class file hierarchy in the package that represents the enhancement. It does not need to match the package of the enhanced type.

Syntax for Defining Enhancements

Although using enhanced properties and methods is straightforward, a special syntax is necessary for defining new enhancements. Defining a new Gosu enhancement looks similar to defining a new Gosu class, with some minor differences in their basic definition.

Differences between classes and enhancements include

- Use the keyword `enhancement` instead of `class`
- To define what to enhance, use the syntax: "`:` *`TYPETOEXTEND`*" instead of "`extends` *`CLASSTOEXTEND`*"
- If you must reference methods on the enhanced class/type, use the symbol `this` to see the enhanced class/type. For example, to call the enhanced object's `myAction` method, use the syntax `this.myAction()`. In contrast, never use the keyword `super` in an enhancement.

  Note: Enhancements technically are defined in terms of the *external interface* of the enhanced type. The keyword `super` implies a superclass rather than an interface, so it is inappropriate for enhancements.

- Enhancements cannot save state information by allocating new variables or properties on the enhanced type.

Enhancement methods can use properties already defined on the enhanced object or call other enhanced methods.

You can add new *properties* as necessary and access the properties on the class/type within Gosu. However, that does not actually allow you to save state information for the enhancement unless you can do so using variables or properties that already exist on the enhanced type. See later in this section for more on this topic.

For example, the following enhancement adds one standard method to the basic `String` class and one property:

```
package example

enhancement StringTestEnhancement : java.lang.String {

  public function myMethod(): String {
    return "Secret message!"
  }

  public property get myProperty() : String {
    return "length : " + this.length()
  }
}
```

Note the use of the syntax "`property get`" for the method defined as a property.

With this example, use code like the following to get values:

```
// get an enhancement property:
print("This is my string".myProperty)

// get an enhancement method:
print("This is my string".myMethod())
```

These lines outputs the following:

```
"length: 17"
"Secret message!"
```

Enhanced methods can call other methods internally, as demonstrated with the `getPrettyLengthString` method, which calls the built-in `String` method `length()`.

IMPORTANT Enhancements can create new methods but cannot override existing methods.

Setting Properties in Enhancements

Within enhancement methods, your code can set other values as appropriate such as an existing class instance variable. You can also set properties with the "`property set` *`PROPERTYNAME`*`()`" syntax. For example, this enhancement creates a new settable property that appends an item to a list:

```
package example

enhancement ListTestEnhancement<T> : java.util.ArrayList<T>
{
  public property set LastItem(item : T) {
    this.add(item)
  }
}
```

Test this code in the Gosu Tester with this code:

```
uses java.util.ArrayList

var strlist = new ArrayList<String>() {"abc", "def", "ghi", "jkl"}

print(strlist)
strlist.LastItem = "hello"
print(strlist)
```

This code outputs:

```
[abc, def, ghi, jkl]
[abc, def, ghi, jkl, hello]
```

You can add new properties and add property set functions to set those properties. However, in contrast to a class, enhancements cannot define **new** variables on the type to store instance data for your enhancement. This limits most types of state management if you cannot directly change the source code for the enhanced type to add more variables to the enhanced type. Enhancements cannot add new variables because different types have dramatically different property storage techniques, such as a persistent database storage, Gosu memory storage, or file-based storage. Enhancements cannot transparently mirror these storage mechanisms.

Also, although enhancements can add properties, enhancements cannot override existing properties.

IMPORTANT Enhancements can add new properties by adding new dynamic property get and set functions to the type. However, enhancements cannot override property get or set functions. Also, enhancements cannot create new native variables on the object that would require additional data storage with the original object. Enhancements cannot override methods either.

Enhancement Naming and Package Conventions

The name of your enhancement must follow the following naming convention of the enhanced type name, then an optional functional description, followed by word `Enhancement`. In other words, the format is:

```
[EnhancedTypeName][OptionalFunctionalDescripton]Enhancement
```

For example, to enhance the `Report` class, you could call it simply:

```
ReportEnhancement
```

If the enhancement added methods related to claim financials, you might emphasize the enhancement's functional purpose by naming the enhancement:

```
ReportFinancialsEnhancement
```

Enhancement Packages

Use your own company package to hierarchically group your own code and separate it from built-in types, in almost all cases. For example, you could define your enhancement with the fully-qualified name `com.mycompany.ReportEnhancement`. Even if you are enhancing a built-in type, if at all possible use your own package for the enhancement class itself.

In only extremely rare cases, you might need to enhance a built-in type and you need to use a `protected` property or method. If so, you might need to define your enhancement in a subpackage of the enhanced type. See "Modifiers" on page 135 for more information about the `protected` keyword. However, to avoid namespace conflicts with built-in types, avoid this approach if possible.

Enhancements on Arrays

To specify the enhanced type for an enhancement on an array type:

- For regular types, use standard array syntax, such as `String[]`.
- For generic types, use the syntax `T[]`, which effectively means all arrays.

chapter 15

Gosu Blocks

Gosu blocks are a special type of function that you can define in-line within another function. You can then pass that block of code to yet other functions to invoke as appropriate. Blocks are very useful for generalizing algorithms and simplifying interfaces to certain APIs. For example, blocks can simplify tasks related to collections, such as finding items within or iterating across all items in a collection.

This topic includes:

What Are Blocks?

Gosu blocks are functions without names (sometimes called *anonymous functions*) that you can define in-line within another function. You can then pass that block of code to yet other functions to invoke as appropriate. Blocks can be very useful for generalizing algorithms and simplifying interfaces to APIs. An API author can design most of an algorithm but let the API consumer contribute short blocks of code to complete the task. The API can use this block of code and call it once or possibly many times with different arguments.

For example, you might want to find items within a collection that meet some criteria, or to sort a collection of objects by certain properties. If you can describe your find or sort criteria using small amount of Gosu code, Gosu takes care of the general algorithm such as sorting the collection.

Some other programming languages have similar features and call them *closures* or *lambda expressions*. For those who use the Java language, notice that Gosu blocks serve some most common uses of single-method anonymous classes in Java. However, Gosu blocks provide a concise and clear syntax that makes this feature more convenient in typical cases.

Blocks are particularly valuable for the following:

- **Collection manipulation.** Using collection functions such as `map` and `each` with Gosu blocks allows concise easy-to-understand code with powerful and useful behaviors for real-world programming.
- **Callbacks.** For APIs that wish to use callback functions after an action is complete, blocks provide a straightforward mechanism for triggering the callback code.
- **Resource control.** Blocks can be useful for encapsulating code related to connection management or transaction management.

Gosu code using blocks appropriately can simplify and reduce the size of your Gosu code. However, they can also be confusing if used too aggressively and use them carefully. If your intended use does not fall into one of the list categories, reconsider whether to use blocks. There may be a better and more conventional way to solve the problem. Generally speaking, if you write a method that takes more than one block as a function/method argument, strongly consider redesigning or refactoring the code.

> **WARNING** Gosu blocks are not always the correct design solution. For example, if you design a function that takes more than one block as arguments, a general rule is to redesign or refactor your code.

Basic Block Definition and Invocation

To define a Gosu block, type use the backslash character (\) followed by a series of arguments. The arguments must be name/type pairs separated by a colon character (`:`) just as if defining arguments in a method. Next, add a hyphen character (`-`) and a greater-than character (`>`) to form the arrow-like pair characters `->`. Finally, add a Gosu expression or a statement list surrounded by curly braces.

In other words, the syntax is:

```
\ argumentList -> blockBody
```

The argument list (*`argumentList`*) is a standard function argument list, for example:

```
x : Number, y : Number
```

The argument list defines what parameters must be passed to the block. The parameter list uses identical syntax as parameters to regular functions. However, in some cases you can omit the types of the parameters, such as passing a block directly into a class method such that the parameter type can be inferred. For examples, see “Argument Type Inference Shortcut In Certain Cases” on page 169.

The block body (*`blockBody`*) can be either of the following:

- a simple expression. This includes anything legal on the **right-hand** side of an assignment statement. For example, the following is a simple expression:

  ```
  "a concatenated string " + "is a simple expression"
  ```

- a statement list with one or more statements surrounded by braces and separated by semi-colon characters, such as the following simple one-statement statement list:

  ```
  \ x : Number, y : Number -> { return x + y }
  ```

  For single-statement statement lists, you *must* explicitly include the brace characters. In particular, note that variable assignment operations are always statements not expressions. Thus, the following expression is invalid:

  ```
  names.each( \ n -> myValue += n )
  ```

  Instead, change it to the following:

  ```
  names.each( \ n -> { myValue += n } )
  ```

  For multiple statements, separate the statements with a semi-colon character. For example:

  ```
  \ x : Number, y : Number -> { var i = x + y; return i }
  ```

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

```
var square = \ x : Number-> x * x    //no need for braces here
var myResult = square(10)         // call the block
```

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

IMPORTANT All parameter names in a block definition's argument list must not conflict with any existing in-scope variables, including but not limited to local variables.

The Gosu editor displays a block definition's backslash character as a Greek lambda character. This improves code appearance and honors the theoretical framework from which blocks derive, called *lambda calculus*. The Gosu editor displays the pair of characters -> as an arrow symbol.

For example, you could type the following Gosu block:

```
var square = \ x : Number -> x * x
```

The Gosu editor displays it as:

```
var square = λ x : Number → x
```

In general, the standard Gosu style is to omit all semicolon characters in Gosu at the end of lines. Gosu code is more readable without optional semicolons. However, if you provide statement lists on one line, such as within block definitions, use semicolons to separate statements. For other style guidelines, see "General Coding Guidelines" on page 279.

Return Values and Return Type

Notice that the block definition does not explicitly declare the *return type*, which is the type of the return value of the block. This is because the return type is inferred from either the expression (if you defined the block with an expression) or for statement list by examining the `return` statements. This frees you of the burden of explicitly typing the return type. This also allows the block to appear short and elegant. However, it is important to understand that the return type is actually *statically typed* even though the type is not explicitly visible in the code.

For example, note the following simple block:

```
var blockWithStatementBody = \ -> { return "hello blocks" }
```

Because the statement `return "hello blocks"` returns a `String`, that means the block's return type is `String`.

IMPORTANT Gosu infers a block's return type by the returned value of the return statements of the statement list. If an expression is provided instead of a statement list, Gosu uses the type of the expression. That type is static (fixed) at compile time although it is not explicitly visible in the code.

Using and Invoking Blocks

Blocks are invoked just like normal functions by referencing a variable to which you previously assigned the block. To use a block, type:

1. the name of the block variable or an expression that resolves to a block
2. an open parenthesis
3. a series of argument expressions
4. a closing parenthesis

For example, suppose you create a Gosu block with no arguments and a simple return statement:

```
var blockWithStatementBody = \-> { return "hello blocks" }
```

Because the statement list returns a `String`, Gosu infers that the block returns a `String`. The new block is assigned to a new variable `blockWithStatementBody`, and the block has a return type of `String` even though this fact is not explicit in the code text.

To call this block and assign the result to variable `myresult`, simply use this code:

```
var myresult = blockWithStatementBody()
```

The value of the variable `myresult` is the `String` value `"hello blocks"` after this line executes:

The following example creates a simple block that adds two numbers as parameters and returns the result:

```
var adder = \ x : Number, y : Number -> { return x + y }
```

After defining this block, you can call it with code such as:

```
var mysum = adder(10, 20)
```

The variable `mysum` has the type `Number` and has the value 30 after the line is executed.

You can also implement the same block behavior by using an expression rather than a statement list, which allows an even more concise syntax:

```
var adder = \ x : Number, y : Number -> x + y
```

Variable Scope and Capturing Variables In Blocks

Gosu blocks maintain some context with respect to the enclosing statement in which they were created. If code in the block refers to variables that are defined outside the scope of the block's definition but in scope where the block is defined, the variable is *captured*. The variable is incorporated **by reference** into the block. Incorporating the variable by reference means that blocks do not merely capture the current value of the variable at the time its enclosing code creates the block. If the variable changes after the enclosing code creates the block, the block gets or sets the most recent value in the *original* scope. This is true even if the *original* scope exited (finished).

The following example adds 10 to a value. However, the value 10 was captured in a local variable, rather than included in an argument. The captured variable (called `captured` in this example) is used but not defined within the block:

```
var captured = 10
var addTo10 = \ x : Number -> captured + x
var myresult = addTo10(10)
```

After the third line is executed, `myresult` contains the value 20.

A block captures the state of the stack at the point of its declaration, including all variables and the special symbol `this`, which represents the current object. For example, the current instance of a Gosu class running a method.

This capturing feature allows the block to access variables in scope *at its definition*:

- ...even after being passed as an argument to another function
- ...even after the block returns to the function that defines it
- ...even if some code assigns it to a variable and keeps it around indefinitely
- ...even after the original scope exits (after it finishes)

In other words, each time the block runs, it can access all variables declared in that original scope in which it was defined. The block can get or set those variables. The values of captured variables are evaluated each time the block is executed, and can be read or set as desired. Captured variable values are **not** simply a static snapshot of their value at the time the block was created.

To illustrate this point further, the following example creates a block that captures a variable (`x`) from the surrounding scope. Next, the code that created the block changes the value of `x`. Only after that change does any code actually call the block:

```
// define a local variable, which is captured by a block
var x = 10

// create the block
var captureX = \ y : Number -> x + y

// Note: the variable "x" is now SHARED by the block and the surrounding context
```

```
// Now change the value of "x"
x = 20

// at the time the block runs, it uses the current value of x,
// this is NOT a snapshot of what it was at the time block was created
var z = captureX( 10 )

print(z) // prints 30   --- not 20!!!
```

The captured variable is effectively **shared** by the original scope and the block that was created within that scope. In other words, the block references the variable itself, not merely its original value.

IMPORTANT If accessing variables not defined within the block definition, blocks effectively share the variable with the context that created it. This is true even if the original scope exited (finished) or its value has changed. This is a very powerful feature. If you use this feature at all, use it very carefully and document your assumptions so people who read your code can understand and debug it.

Argument Type Inference Shortcut In Certain Cases

The Gosu parser provides additional type inference in a common case. If a block is defined within a method call parameter list, Gosu can infer the type of the block's arguments from the parameter argument. You do not need to explicitly specify the argument type in this case.

In other words, if you pass a block to a method, in some cases Gosu can infer the type so you can omit it for more concise code. This is particularly relevant for using collection-related code that takes blocks as arguments.

For example, suppose you had this code:

```
var x = new ArrayList<String>(){"a", "b", "c"}

var y = x.map( \ str : String -> str.length )
```

You could instead omit the argument type (`String`). The `map` method signature allows Gosu to infer the argument type in the block because of how the `map` method is defined.

You could use the more concise code:

```
var x = new ArrayList<String>(){"a", "b", "c"}

var y = x.map( \ str -> str.length )
```

The list method `map()` is a built-in list enhancement method that takes a block with one argument. That argument is always the same as the list's type. Therefore Gosu infers that `str` must be of type `String` and the you do not need to explicitly define the type of arguments nor the return type.

Note: The `map` method is implemented using a built-in Gosu enhancement of the Java language `List` class. For more information, see "Collections" on page 183.

Block Type Literals

Block literals are a form of type literal, which means the way you reference a *block type*. The block literal specifically what kinds of arguments the block takes and what type of return value it returns.

Block Types In Declarations

If you define a variable to contain a block in a variable declaration, the preferred syntax is:

```
variableName( list_of_types ) : return_type
```

For example, to declare that `x` is a variable that can contain a block that takes a single `String` argument and returns a `String` value, use this code:

```
var x( String ) : String
```

In declarations, you can also optionally use the `block` keyword, although this is discouraged in declarations:

```
block( list_of_types ) : return_type
```

For example, this code declares the same block type as described earlier:

```
var x : block( String ) : String
```

Block Types Not Part of Declarations

Where a block type literal is **not** part of a declaration, the `block` keyword is strictly required:

```
block( list_of_types ) : return_type
```

For example:

```
var b = block( String ) : Number
```

This means that the `b` variable is **assigned** a value that is a block type. Since the block type literal is not directly part of the declaration, the `block` keyword must be specified.

Block Types In Argument Lists

Within function definition, a function argument can be a block. As you define the block argument, provide a **name** for that block parameter so you can use it within the function. Do this using the following syntax for block types in argument lists:

```
parameter_variable_name( list_of_types ) : return_type
```

For example, suppose you want to declare a function that took one argument, which is a block. Suppose the block takes a single `String` argument and returns no value. If you want refer to this block by name as `myCallback`, define the argument using the syntax:

```
myCallBack( String ) : void
```

It might be easier to understand with an actual example. The following Gosu class includes a function that takes a callback block. The argument is called *`myCallBack`*, which is a block that takes a single string argument and returns no value. The outer function calls that callback function with a `String`.

```
package mytest

class test1 {
  function myMethod( myCallBack( String ) : void ) {

    // call your callback block and pass it a String argument
    myCallBack("Hello World")

  }
}
```

Test this code as follows:

```
var a = new mytest.test1()
a.myMethod( \ s : String -> print("<contents>" + s + "</contents>") )
```

For even more concise code, you can omit the argument type “`: String`” in the in-line block. The block is defined in-line as an argument to a method whose argument types are already defined. In other words, you can simply use the following code

```
var a = new mytest.test1()
a.myMethod( \ s -> print("<contents>" + s + "</contents>") )
```

Both versions print the following:

```
<contents>Hello World</contents>
```

Block Types BNF Notation

For those interested in formal BNF notation, the notation of a block literal is:

```
blockliteral -> block_literal_1 | block_literal_2
block_literal_1 -> block ( type_list ) : type
block_literal_2 -> parameter_name ( type_list ) : return_type
type_list -> type | type_list , | null
```

Blocks and Collections

Gosu blocks are particularly valuable for working with collections of objects. Blocks allow concise and easy-to-understand code that loops across items, extracts information from every item in a collection, or sorts items. Common collection enhancement methods that use blocks are `map`, `each`, and `sortby`.

For example, suppose you want to sort the following list of strings:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}
```

You could easily resort the list based on the length of the strings using blocks. Create a block that takes a `String` and returns the sort key, which in this case is the string's length. The built-in list `sortBy(...)` method handles the rest of the sorting algorithm and then returns the new sorted array:

```
var resortedStrings = myStrings.sortBy( \ str -> str.Length() )
```

These block-based collection methods are implemented using a built-in Gosu enhancement of the Java language `List` class. For more information, see “Collections” on page 183.

Blocks as Shortcuts for Anonymous Classes

You can use a block in place of an anonymous class that implements an interface with a single method. For more information, see “Classes” on page 127.

chapter 16

Gosu Generics

Gosu generics is a language feature that lets you define a class or function as working with many types by abstracting its behavior across multiple types of objects. This abstraction feature is important because collections defined with generics can specify what kinds of objects they contain. If you use collections, you can be specific about what objects are in the collection. You do not need to very general about the type of the contents, such as using a root type such as `Object`. However, if designing APIs that can work with different types of objects, you can write the code only once and it works with different types of collections. In essence, you can generalize class or methods to work with various types and retain compile-time type safety. Use generics to write statically typed code that can be abstracted to work with multiple types.

Generics are especially valuable for defining special relationships between arguments to a function and/or its return values. For example, you can require two arguments to a function to be homogenous collections of the same type of object and the function returns the same type of collection. Designing APIs to be abstract like that allows your code and the Gosu language to infer other relationships. For example, an API that returns the first item in a collection of `String` objects is always typed as a `String`. You need not write coercion code with the syntax "`as` *`TYPE`*" as often if designing your APIs to use generics. Because generics increase how often Gosu can use *type inference*, your collection-related code can be easy-to-understand, concise, and type-safe.

Gosu generics are compatible with generics in Java version 1.5, so you can use Java classes designed for Java 1.5 generics or even extend them in Gosu.

For more information about static typing in Gosu, see "More About the Gosu Type System" on page 29.

This topic includes:

Gosu Generics Overview

You probably use simple arrays sometimes to store multiple objects of the same type. For example, an array of five numbers, an array of forty-seven `String` objects, or an array of some other type of primitive or object. Similarly, *collections* (including all *lists*) provide another way of grouping items together but with important differences between arrays and collections.

Standard arrays contain items of the same type of object and if one type extends another, you can make certain assumptions about the type of items in the array. For example, because `Integer` extends `Number`, it means that an array of `Integer` is also an array of `Number`. In other words, `Integer[]` is also an array of `Number[]`. Where a `Number[]` is required**,** you are free to pass or assign an `Integer[]`. However, collections do not work that way.

Standard collections can contain a variety of types of objects (they are *heterogeneous*). If you take an object out of a collection, typically you must cast it to a desired type or check its type before doing something useful with it.

Without generics, in practice, people tend to design collection-related APIs to work with collections of `Object` instances. The `Object` class is the root class of all non-primitive objects in the Gosu language and also in the Java language.

Unfortunately, if you use APIs that return collections of type `Object`, your code must cast (coerce) it to a more specific type to access properties and methods.

Although casting one value to another type is useful, it is unsafe in some cases and prevents the compiler from knowing at compile-time whether it succeeds. For example, if the item you extract from the collection is a type that does not support casting, it fails at **run time**. For example, casting a `String` to an `Array`. This approach to coding is inconsistent with confirming type problems at **compile time**. Detecting problems at compile time is important for reliable mission-critical server applications.

A simple alternative is to define different types of collections/lists that support only homogenous sets of objects of a certain type, such as a `StringList`, an `IntegerList`, or a `MyCustomClassList`. This provides compile-time certainty and thus dramatically reduces the chance of type safety issues. However, the downside is more complexity to make the API work with different types of lists. A Gosu class method that takes a `StringList` would need a separate method signature to take an `IntegerList`. This type of repetitive method signature declaration simply to achieve type safety is time consuming. Additionally, it might be incomplete: if you provide an API, it cannot predict list of types you do not know about that a consumer of your API wants to use. If only there were a way to generalize the function so that it would work with all lists, you could provide a generalized (or generic) function to perform the task.

Suppose you could define a collection with an explicit type of each item. With the angle-bracket notation after the collection class such as `List<Number>`, you can specify what types of things the container contains. If you read aloud, you can translate the bracket notation in English as the word “of”. Thus, in English, the syntax `List<Number>` means “a list of numbers”. Even better, suppose there was a way to define function parameters to work with **any type**. What if it always returns an object of the same type, or an array of that type, and have such relationships enforced at compile time? This is what Gosu generics do for you.

Generics provide a way to specify the **type of objects in a collection** with specificity as you use an API but with generality as you design an API. At compile time, the Gosu compiler confirm everything has a valid type for the API. Additionally, Gosu **infers types** of arguments and return values in many cases so you do not have to do much coercing values from a root class. For instance, you do not generally need to coerce a variable of type `Object` to a more useful specific type. Suppose you take values out of a collection of objects of type `MyClass`. A variable that contains an extracted first item in the collection always has type `MyClass`, not `Object`. With generics you do not need to coerce the value to type `MyClass` before calling methods specific to the `MyClass` class.

Generics provide the best of *generalizability* as you design APIs and *specificity* as you use APIs. Using generics, your collections-related code can be easy-to-understand, concise, and typesafe.

Gosu generics are compatible with generics implemented in Java version 1.5. You can use Java utility classes designed for Java 1.5 generics and even extended them in Gosu. There is one exception for Java-Gosu compatibility, which is that Gosu does not support the syntax `<? super TYPE>`. For more information about other similar features, see “Bounded Wildcards” on page 178.

For extended discussions of generics as implemented in Java, see the book “Java Generics and Collections” by Maurice Naftalin and Philip Wadler, or the following on-line tutorial:

```
http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
```

Gosu Generics are Reified

One important difference between Gosu and Java is that Gosu generics are *reified*. This means that unlike Java, at run time, Gosu retains the actual specific type. In other words, at run time you could check whether an object was an instance of `PetGroup<Cat>` or `PetGroup<Dog>` including the information in the angle brackets.

In contrast, Java generics lose this generic parameter information at run time. This is called *type erasure*. Java introduced generics in this way to maximize compatibility with older Java code that did not support generics.

Using Gosu Generics

If a function or method has already defined arguments or return value using Gosu generics, as a “consumer” of this API, the API is easy to use. The only important thing to know is that you define the type of collection with the angle bracket notation `COLLECTION_CLASS<OF_TYPE>`. For example, an array list of `Address` objects would use the syntax `ArrayList<Address>`.

> **Note:** In practice, you sometimes do not need to define the collection type due to type inference or special object constructors. See “Basic Lists” on page 183 and “Basic HashMaps” on page 185.

For example, suppose you want a list of `String` objects. One way to define the list would be:

```
var theList= new ArrayList<String>() { address1, address2, address3, address4 }
```

You could create a function that takes a specific type of list, in this case a list of strings:

```
function printStrings( strs : ArrayList<String> ) {
  for( s in strs ) {
  print( s )
  }
}
```

If you want to call a method using Gosu generics to take an array list of any type, simply call the method:

```
r = printStrings(theList)
```

If Gosu knows the return result type of a function, it can infer the type of other things, which makes your code more concise:

```
var strs = new ArrayList<String>(){"a", "ab", "abc"}
var longerStrings = strs.findAll( \ str -> str.length >= 2 )
```

In the previous example, the resulting value of `longerStrings` is strongly typed as `ArrayList<String>`. This is because Gosu knows the return type of the `findAll` method called on any array list of `String` values. If you get an object from value of `longerStrings`, it has the correct expected `String` type, not simply `Object`.

Using functions defined with generics typically are simple, often even more simple because of Gosu generics. The return value can be **strongly typed** to match the exact type of collection you passed into it or one of its items. For example, return a “list of `MyClass` objects” or “a `MyClass` object”, rather than a “list of `Object`” or just an `Object`. Although generics are abstract, using APIs other people defined with generics typically is straightforward and intuitive.

The Power of Generics Comes From Wildcards

Although you can specify a specific type of collection, the greatest power of Gosu generics is defining APIs that work with multiple types, not just a single type. This requires a special syntax using wildcards.

Without Gosu generics, the way to support multiple types would be define a utility class method that takes a standard `Collection` object and returns another `Collection` object. That would allow you to use the method with a wide variety of collections: a `Collection` of `MyClass` objects, a `Collection` of `Address` objects, and so on. However, any code that extracted items from the collection after an API call would have to add code with a coercion "`x as TYPE`" if extracting an object from it:

```
var f = myCollection.iterator.next()
(f as MyClass).myClassMethod()
```

Note the code "`f as MyClass`". That approach typically results in hard-to-read code, since you must manually add casting to a specific type for a variety of APIs due to this issue. Additionally, it is dangerous because the actual casting happens at run time and you could make a mistake by casting it to the wrong object. Most importantly, the casting could **fail at run time** in some cases if you make other types of errors, rather than identified and flagged at compile time. Fortunately, with generics this type of casting is not necessary if you use APIs designed with generics and design any new APIs with generics.

That would allow us to remove the cast (the "`as MyClass`") from the previous example:

```
var a = new ArrayList<MyClass>() { c1, c2, c3 }
...

// the result of this is strongly typed
var first = myResults.iterator.next().myClassMethod
```

From quickly looking at the code, you might assume from the text that the `first` variable is not strongly typed after removing the cast. However, it is strongly typed at compile time.

If you want to make full use of the language's ability to use generic types, you have two choices:

- *parameterize* a class, which means to add generic types to the class definition
- *parameterize* a method, which means to add generic types to a method definition

Parameterized Classes

If you want a class that always operates with a generic type, define the class with the angle bracket notation `CLASSNAME<GENERIC_TYPE_NAME>` in the class definition. By convention, for the *`GENERIC_TYPE_NAME`* string, use a one-letter variable, preferably `T`. For example, you could define a class `MyClass` as `MyClass<T>`.

In the following example, the class `Genericstest` has one method that returns the first item in a list. Gosu strongly types the return value to match the type of the items in the collection:

```
package com.example
uses java.util.ArrayList

class Genericstest<T>
{
  // print out (for debugging) and then return the first item in the list, strongly typed
  public function PrintAndReturnFirst(aList : ArrayList<T>) : T {
    print(aList[0])
    return aList[0]
  }
}
```

Now, some other code could use this class and pass it an array list of any type:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}

var t = new Genericstest<String>()
var first = t.PrintAndReturnFirst( myStrings )
```

After this code runs, the value of the variable `first` is **strongly typed** as `String` because of how it used the method that was defined with generics.

This also works with multiple dimensions of types. Suppose you want to write something that stores key-value maps. Instead of writing:

```
class Mymapping {
  function put( key : Object, value : Object) {...}
  function get( key : Object) : Object {...}
}
```

...you could use generics to define it as:

```
class Mymapping<K,V> {
  function put( key : K, value : V) {...}
```

```
  function get( key : K) : V {...}
}
```

Now you can use this class with strongly typed results:

```
myMap = new Mymapping<String, Integer>
myMap.put("ABC", 29)

theValue = myMap.get("ABC")
```

The `theValue` variable is strongly typed at compile time as `Integer`, not `Object`.

Within the method definition, the values in angle brackets have special meanings as type names in a parameterized class definition. In this case, the `K` and `V` symbols. Use these symbols in method signatures in that class to represent **types** in arguments, return values, and even Gosu code inside the method.

You can think about it as at the time the method runs, the symbols `K` and `V` are *pinned* (assigned) to specific types already. By the time this method runs, some other code created new instances of the parameterized class with specific types already. The compiler can tell which method to call and what types `K` and `V` really represent. In the earlier example, the concrete types are `String` (for `K`) and `Integer` (for `V`).

Gosu generics offer this power to define APIs once and abstract the behavior across multiple types. Define your APIs with the generics and wildcards to generalize your APIs to work with different types of types or collections. Your code is strongly-typed code at compile time, which improves reliability of code at run time.

Parameterized Methods

You can add a finer granularity of type usage by adding the generic type modifiers to the method, immediately after the method name. This is called *parameterizing* the method, or making a *polymorphic method* with a generic type.

For example, in the following example, the class is not parameterized but one method is:

```
package com.example
uses java.util.ArrayList

class Test3
{
  // return the last item in the list
  public function ReturnLast<T>(a : ArrayList<T>) : T{

    var lastItemIndex = a.size - 1
    return a[lastItemIndex]
  }
}
```

Within the method's Gosu code, the symbol `T` can be used as a type and this code works automatically, matching `T` to the type of the collection passed into it.

Code can use this class:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "123"}

var t = new com.example.Test3()

var last = t.ReturnLast( myStrings )

print("last item is: " + last)
```

The variable `last` is strongly typed as `String`, not `Object`.

Other Unbounded Generics Wildcards

In some cases, there is no prior reference to a type wildcard character (such as `T` in earlier examples) if you need to define arguments to a method. This is typical for defining blocks, which are anonymous functions defined in-line within another function (see "Gosu Blocks" on page 165). In such cases, you can simply use the question mark character instead of a letter:

```
var getFirstItem = \ aList : List<?> -> aList[0]
```

For more details about how generics interact with blocks, see “Generics and Blocks” on page 178.

Bounded Wildcards

You can specify advanced types of wildcards if you want to define arguments that work with many types of collections. However, you can still make some types of assumptions about the object’s type. For example, you might want to support *homogenous collections* (all items are of the same type) or perhaps only instances of a class and its subclasses or subinterfaces.

Suppose you had a custom class Shape. Suppose you want a method to work with collections of circle shapes or collections of line shapes, where both Circle and Line classes extend the Shape class. For the sake of this example, assume the collections are always homogenous and never contain a mix of both types.

It might seem like you could define a function like this:

```
public function DrawMe (circleArray : ArrayList<Shape>)
```

The function would work if you pass it an object of type ArrayList<Shape>. However, it would not work if you tried to pass it an ArrayList<Circle>, even though Circle is a subclass of Shape.

Instead, specify support of multiple types of collections while limiting support only to certain types and types that extend those types. Use the syntax “extends *TYPE*” after the wildcard character, such as:

```
<T extends TYPE>
```

or...

```
<? extends TYPE>
```

For example:

```
public function DrawMe (circleArray : ArrayList<T extends Shape>)
```

In English, you can read that argument definition as “the parameter circleArray is an ArrayList containing objects all of the same type, and that type extends the Shape class”.

Although Gosu generics work very similar to generics in the Java language, one other type of bounded wildcard supported by Java is not supported in Gosu. The supertype bounded wildcard syntax <? super *TYPE*> is supported by Java but not by Gosu.

WARNING Gosu does **not** support the generics syntax for bounded supertypes <? super *TYPE*>, which is supported by Java. That syntax is rarely used anyway because the <? extends *TYPE*> is more appropriate for typical code.

Generics and Blocks

The Gosu generics feature is often used in conjunction with another Gosu feature called blocks, which are anonymous functions that can be passed around as objects to other functions. You can use generics to describe or use blocks in two basic ways.

Blocks Can Have Arguments Defined With Generics

You can create a block with arguments and return values that work like the earlier-described function definitions defined with generics. Your block can support multiple types of collections and return the same type of collection passed into it. Use a question mark (?) wildcard symbol to represent the type, such as ArrayList<?>.

Note: In block definitions you cannot use a letter as a wildcard symbol, such as ArrayList<T>. Gosu only supports the letter syntax for parameterized classes and methods.

The following example uses the <?> syntax to define an ArrayList using generics:

```
uses java.util.ArrayList

// set up some sample data in a string list
var s = new ArrayList<String>() {"one", "two", "three" }
```

```
// define a block that gets the first item from a list
var getFirstItem = \ aList : List<?> -> aList[0]

// call your block. notice that the variable is strongly typed as String, not as Object
var first = getFirstItem(s)

print(first)
```

This code prints the value:

```
one
```

Notice that the return result is strongly typed and Gosu infers the appropriate type from the block.

Functions that Take Blocks as Arguments

Also, there is a more complex type of interaction between blocks and generics. You can pass blocks as objects to other functions. If a function takes a **block as an argument**, you can define that function argument using generics to abstractly describe the appropriate set of acceptable blocks.

To answer questions like “what kind of block does this function support?”, determine the number of arguments, the argument types, and the return type. For example, consider a block that takes a `String` and returns another `String`. The type definition of the block itself indicates one argument, the parameter type `String`, and the return type `String`.

If you want to support a wide variety of types or collections of various types, define the block using generics. If you define your APIs this way, you permit consumers of your APIs to it with a wide variety of types and use strong typing and type inference.

If a class method on a parameterized class (a class using generics) takes a block as an argument, Gosu uses the types of the arguments. You can **omit** the type of the arguments as you define the block.

A typical example of this is the list method `sortBy,` which takes a block. That block takes exactly one argument, which must be the same type as the items in the list. For example, if the list is `ArrayList<String>`, the block must be a `String`. The method is defined as an *enhancement* with the following signature:

```
enhancement GWBaseListEnhancement<T> : java.util.List<T>
...
  public function sortBy( value(T):Comparable ) : java.util.List<T>
```

Note the use of the letter T in the enhancement definition and in the method signature:

```
value(T):Comparable
```

That syntax means that the argument is a block that takes one argument of type T and returns a Comparable value (such as an `Integer` or `String`).

Suppose you had an array list of strings:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}
```

You could easily resort the list based on the length of the strings using blocks. Create a block that takes a `String` and returns the sort key, in this case the text length. Let the `List.sortBy(...)` method handle the rest of the sorting algorithm details and return the new sorted array.

```
var resortedStrings = myStrings.sortBy( \ str -> str.Length() as Integer )
```

It is important to understand that this example omitted the type of the block argument `str`. You do not have to type:

```
var resortedStrings = myStrings.sortBy( \ str : String -> str.Length() as Integer )
```

Type inference in cases like this valuable for easy-to-understand and concise Gosu code that uses generics.

IMPORTANT If you define a block as an argument to a method, you can omit the argument types in the block in some cases. Omit the type if Gosu can infer the type from the arguments required of that method. Omitting the type in cases in which you can do so leads to concise easy-to-read code.

Practical examples of this approach, including the method definitions of the built-in `sortBy` method are shown in the following section, “How Generics Help Define Collection APIs” on page 180.

For extensive information about similar APIs with blocks, see “Gosu Blocks” on page 165. For specific examples of built-in APIs that use generics with blocks, see “Collections” on page 183.

How Generics Help Define Collection APIs

By using Gosu generics to define function parameters, you can enforce type safety yet make logical assumptions about interaction between different APIs. This is most notable the Gosu feature called *blocks*, which allows in-line creation of anonymous functions that you can pass to other APIs.

For example, you could easily resort a list of `String` objects based on the length of the strings using these two features combined:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}
var resortedStrings = myStrings.sortBy( \ str -> str.length as Integer)
```

If you want to print the contents, you could print them with:

```
resortedStrings.each( \ str  -> print( str )  )
```

...which would produce the output:

```
a
ab
abc
abcd
```

This concise syntax is possible because the `sortBy` method is defined a single time with Gosu generics.

It uses the wildcard features of Gosu generics to work with **all** lists of type `T`, where `T` could be any type of object, not just built-in types. The method is defined as a Gosu enhancement to all `List` objects. This means that the method automatically works with all Java objects of that class from Gosu code, although the method is not defined in Java. Enhancement definitions look similar to classes. The enhancement for the `sortBy` method looks like:

```
enhancement GWBaseListEnhancement<T> : java.util.List<T>
  ...
  ...
  public function sortBy( value(T):Comparable ) : java.util.List<T> {
    ...
  }
}
```

That means that it works with all lists of type `T`, and the symbol `T` is treated as the type of the collection. Consequently, the `sortBy` method uses the type of collection (in the earlier example, an array list of `String` objects). If the collection is a list of `String` objects, method must takes a comparison function (a *block*) that takes a `String` object as an argument and returns a `Comparable` object. The symbol `T` is used again in the return result, which is a list that has the same type passed into it.

IMPORTANT For a reference of extremely powerful collection-related APIs that use blocks and Gosu generics, see “Collections” on page 183

Multiple Dimensionality Generics

Typical use of generics is with one-dimensional objects, such as lists of a certain type of object, such as a list of `String` objects, or a collection of `Address` objects. However, generics are flexible in Gosu as well as Java to include multiple dimensionality.

For example, a `Map` stores a set of key/value pairs. Depending on what kind of information you are storing in the `Map`, it may be useful to define APIs that work with certain types of maps. For example, maps that have keys that have type Long, and values that have type String. In some sense, a Map is a two-dimensional collection, and you can define a map to have a specific type:

```
Map<Long, String> contacts = new HashMap<Long, String>()
```

Suppose you want to define an API that worked with multiple types of maps. However, the API would return a value from the map and it would be ideal if the return value was strongly typed based on the type of the map. You could use a 2-dimensional generics with wildcards, to define the method signature:

```
public function GetHighestValue( themap : Map<K,V>) : V
```

The argument themap has type Map and specifies two type wildcards (single capital letter) separated by commas. In this case, assume the first one represents the type of the key (K) and the second one represents the type of the value (V). Because it uses the V again in the return value type, the Gosu compiler makes assumptions about relationships between the type of map passed in and the return value.

For example, suppose you pass the earlier example map of type <Long, String> to this API. The compiler knows that the method returns a String value. It knows this because of the two uses of V in the method signature, both as parameter and as return value.

Generics With Custom ‘Containers’

Although Gosu generics are most useful with collections and lists, there is no requirement to use these features with built-in Collection and List classes. Anything that metaphorically represents a “container” for other objects might be appropriate for using Gosu generics to define the type of items in the container.

Abstract Example

Suppose you want to write something that stores key-value maps. Instead of writing:

```
class Mymapping {
  function put( key : Object, value : Object) {...}
  function get( key : Object) : Object {...}
}
```

...you could use generics to define it as:

```
class Mymapping<K,V> {
  function put( key : K, value : V) {...}
  function get( key : K) : V {...}
}
```

Now you can use this class with strongly typed results:

```
myMap = new Mymapping<String, Integer>
myMap.put("ABC", 29)

theValue = myMap.get("ABC")
```

The theValue variable is strongly typed at compile time as Integer.

Real-world Example

Suppose you were writing a program for an automotive manufacturing company and want to track vehicles within different factories during production. Suppose you want to represent cars with a Car object, trucks with a Truck object, vans with a Van object, and these all derive from a root class Vehicle.

You could create some sort of custom container object called Factory that does not derive from the built-in collection classes. For the purpose of this example, assume that each factory only contains one type of vehicle. A FactoryGroup could contain multiple Car objects, or multiple Truck objects, or multiple Van objects.

Suppose you need APIs to work with all of the following types:

- a FactoryGroup containing one or more Car objects
- a FactoryGroup containing one or more Truck objects
- a FactoryGroup containing one or more Van objects

You could represent these types of collections using the syntax:

- `FactoryGroup<Car>`
- `FactoryGroup<Truck>`
- `FactoryGroup<Van>`

Perhaps you want an API that returns all vehicles in the last step in a multi-step manufacturing process. You could define the API could be defined as:

```
public function GetLastStepVehicles(groupofvehicles FactoryGroup<T>) : FactoryGroup<T>
```

Because the method uses generics, it works with all types of `FactoryGroup` objects. Because both the same letter `T` appears more than once in the method signature, this defines parallelism that tells Gosu about relationships between arguments and/or return values.

The definition of this method could be understood in English as:

> "The method `GetLastStepVehicles` takes one argument that is a factory group containing any one vehicle type. It returns another factory group that is guaranteed to contain the identical type of vehicles as passed into the method."

Alternatively, you could define your API with bounded wildcards for the type:

```
public function GetLastStepVehicles(groupofvehicles FactoryGroup<? extends Vehicle>) : FactoryGroup<T>
```

Using this approach might allow your code to make more assumptions about the type of objects in the collection. It also prevents some coding errors, such as accidentally passing `FactoryGroup<String>` or `FactoryGroup<Integer>`, which fail at compile time. You can find out about your coding errors quickly.

If you want to make code like this, you also need to tell the Gosu compiler that your class is a container class that supports generics. Simply add the bracket notation in the definition of the class, and use a capital letter to represent the type of the class. For example, instead of typing:

```
public class MyFactory
```

...you would instead define your class as a container class supporting generics using the syntax:

```
public class MyFactory<T>
```

Generics with Non-Containers

There is no technical requirement that you use generics with collections or other containers. However, collections and other containers are the typical uses of generics. You can define any class to use Gosu generics to generalize what it supports or how to work with various types. There is no limit on how you can use generics features for new classes.

For example, suppose you want to generalize a class `MyClass` to work differently with different types.

Do not simply define the class `MyClass` as:

```
public class MyClass
```

Instead, define it as:

```
public class MyClass<T>
```

You also could let your class support multiple dimensions similar to how the `Map` class works with two dimentions. See "Multiple Dimensionality Generics" on page 180. You could define your class abstracted across multiple types, separated by commas:

```
public class MyClass<K, V>
```

chapter 17

Collections

The collection and list classes used frequently in Gosu rely on the Java language's collection classes. However, there are important differences because of built-in *enhancements* to these classes that use *Gosu blocks*, anonymous in-line defined functions that are not directly supported in the Java language. Combining Gosu's enhancement and block features permits concise easy-to-understand Gosu code that manipulates collections. With a single line of code, you can loop across collection items and perform actions on each items, extract information from each item, or sort items.

Related topics:

- "Gosu Blocks" on page 165.
- "Gosu Generics" on page 173.
- "Enhancements" on page 161.

IMPORTANT This topic assumes understanding of blocks and generics. To understand how generics enable the enhancement-based APIs, see "How Generics Help Define Collection APIs" on page 180.

This topic includes:

Basic Lists

Lists in Gosu inherit from the Java interface `java.util.List` and its common subclasses such `java.util.ArrayList`.

Creating a List

To create a list with nothing it, specify the type of object it contains in brackets using *generics notation*, such as in this example using an `ArrayList` of `String` objects:

```
var myemptylist = new ArrayList<String>()
```

For more information about generics, see "Gosu Generics" on page 173.

In many cases you might want to initialize (load) it with data. Gosu has special features that allow a natural syntax for initializing lists similar to initializing arrays.

For example, the following is an example simple array initializer:

```
var s2 = new String[ ] {"This", "is", "a", "test."}
```

In comparison, the following is an example new `ArrayList`:

```
var strs = new ArrayList<String>(){"a", "ab", "abc"}
```

The previous line is effectively shorthand for the following code:

```
var strs = new ArrayList<String>()
strs.add("a")
strs.add("ab")
strs.add("abc")
```

Type Inference and List Initialization

Because of Gosu's intelligent *type inference*, you can use an even more concise initializer syntax for lists:

```
var s3 = {"a", "ab", "abc"}
```

The type of `s3` is `java.util.ArrayList<String>` (a list of String objects) because all list members have the type `String`.

Gosu infers the type of the `List` to be the *least upper bound* of the components of the list. In the simple case above, the type of the variable `x` at compile time is `List<String>`. If you pass different types of objects, Gosu finds the most specific type that includes all of the items in the list.

If the types implement interfaces, Gosu attempts to preserve commonality of interface support in the list type. This ensures your list acts as expected with APIs that rely on support for the interface. In some cases, the resulting type is a *compound type*, which combines a *class* and one or more *interfaces* into a single type. For example, the following code initializes an `int` and a `double`:

```
var s  = {0, 3.4}
```

The resulting type of `s` is `ArrayList<java.lang.Comparable & java.lang.Number>`. This means that it is an array list of the compound type of the class `Number` and the interface `Comparable`.

> **Note:** The `Number` class does not implement the interface `Comparable`. If it did, then the type of `s` would simply be `ArrayList<java.lang.Number>`. However, since it does not implement that interface, but both `int` and `double` implement that interface, Gosu assigns the compound type that includes the interfaces that they have in common.

Also see related section "Compound Types" on page 260.

Getting and Setting List Values

The following verbose code sets and gets `String` values from a list using the native Java `ArrayList` class:

```
var strs = new ArrayList<String>(){"a", "ab", "abc"}
strs.set(0, "b")
var firstStr = strs.get(0)
```

You can write this in Gosu instead in the more natural index syntax using Gosu shortcuts:

```
var strs = {"a", "ab", "abc"}
strs[0] = "b"
var firstStr = strs[0]
```

Gosu does not automatically resize lists using this syntax. If a list has only three items, the following code does not work:

```
strs[3] = "b" // index number 2 is the higest supported number
```

Gosu provides additional initializer syntax for both lists and maps similar to Gosu's compact initializer syntax for arrays.

Special Behavior of 'List' in Gosu

In `new` expressions, you can use the **interface** type `List` rather than the **class** type `ArrayList`. Gosu treats this special case as an attempt to initialize an instance of the class type `ArrayList`.

For example:

```
var strs = new List<String>(){"a", "ab", "abc"}
```

Basic HashMaps

Maps in Gosu inherit from the Java class `java.util.HashMap`.

Creating a HashMap

To create an empty map, specify the type of objects it contains in brackets using *generics notation*. For example, define a `HashMap` that maps a `String` object to another `String` object:

```
var emptyMap = new HashMap<String, String>()
```

For more information about generics, see "Gosu Generics" on page 173.

In many cases you might want to initialize (load) it with data. Gosu has special features that allow a natural syntax for initializing maps similar to initializing arrays and lists.

For example, the following code creates a new `HashMap` where `"a"` and `"c"` are keys, whose values are `"b"` and `"d"` respectively

```
var strMap = new HashMap<String, String>(){"a" -> "b", "c" -> "d"}
```

That is effectively shorthand for the following code:

```
var strs = new HashMap<String, String>()
strs.put("a", "b")
strs.put("c", "d")
```

This syntax makes it easy to declare static final data structures of this type within Gosu, and with easier-to-read code than the equivalent code would be in Java.

Getting and Setting Values in a HashMap

The following code sets and gets `String` values from a `HashMap`:

```
var strs = new HashMap<String, String>(){"a" -> "b", "c" -> "d"}
strs.put("e", "f")
var valueForE = strs.get("e")
```

You can write this instead in the more natural index syntax using Gosu shortcuts:

```
var strs = new HashMap<String, String>(){"a" -> "b", "c" -> "d"}
strs["e"] = "f"
var valueForE = strs["e"]
```

Creating a HashMap with Type Inference

Because of Gosu's intelligent *type inference* features, you can optionally use a more concise initializer syntax if Gosu can *infer* the type of the map.

For example, suppose you create a custom function `printMap` defined as:

```
function printMap( strMap : Map<String, String> ) {
   for( key in strMap.keys ) {
     print( "key : " + key + ", value : " + strMap[key] )
   {
```

```
}
```

Because the type of the map is explicit in the function, callers of this function can use an initializer expression **without** specifying the type name or even the keyword `new`. This does not mean that the list is untyped. The list is statically typed but it is optional to declare explicitly because it is redundant.

For example, you could initialize a `java.util.Map` and call this function with verbose code like:

```
printMap( new Map<String, String>() {"a" -> "b", "c" -> "d"} )
```

Instead, simply type the following code and use type inference for concise code:

```
printMap( {"a" -> "b", "c" -> "d"} )
```

Gosu permits this last example as valid and typesafe. Gosu infers the type of the `List` to be the *least upper bound* of the components of the list. In the simple case above, the type of the variable `x` at compile time is `List<String>`. If you pass different types of objects, Gosu finds the most specific type that includes all of the items in the list.

If the types implement interfaces, Gosu attempts to preserve commonality of interface support in the list type. This ensures your list acts as expected with APIs that rely on support for the interface. In some cases, the resulting type is a *compound type,* which combines a *class* and one or more *interfaces* into a single type. For example, the following code initializes an `int` and a `double`:

```
var s  = {"hello" -> 0, "there" -> 3.4}
```

The resulting type of `s` is `HashMap<String, java.lang.Comparable & java.lang.Number>`. This means that it is a map with two generic parameters:

- `String`
- The compound type of the class `Number` and the interface `Comparable`.

Note: The `Number` class does not implement the interface `Comparable`. If it did, then the type of `s` would simply be `Map<String, java.lang.Number>`. However, since it does not implement that interface, but both `int` and `double` implement that interface, Gosu assigns the compound type that includes the interfaces that they have in common.

Special Enhancements on Maps

Just as most methods for lists are defined as part of Java's class `java.util.ArrayList`, most of the behavior of maps in Gosu inherit behavior from `java.util.Map`. However, Gosu provides additional enhancements to extend maps with additional features, some of which use Gosu blocks.

Map Properties for Keys and Values

Enhancements to the `Map` class add two new read-only properties:

- `keys` - calcuates and returns the set of keys in the `Map`. This is simply a wrapper for the keySet() method.
- `values` - returns the values of the `Map`.

Each Key and Value

Enhancements to the `Map` class add the `eachKeyAndValue` method, which takes a block that has two arguments: of the key type and one of the value type. This method calls this block with **each** key/value pair in the `Map`, allowing for a more natural iteration over the Map.

For example:

```
var strMap = new HashMap<String, String>(){"a" -> "b", "c" -> "d"}
strMap.eachKeyAndValue( \ key, value -> print("key : " + key + ", value : " + value ) )
```

List and Array Expansion (*.)

Gosu includes a special operator for array expansion and list expansion. This array and list expansion can be useful and powerful. The expansion operator is an asterisk followed by a period, for example:

```
names*.Length
```

The return value is as follows:

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

For example, suppose you have an array of `Book` objects, each of which has a `String` property `Name`. You could use array expansion to extract the `Name` property from each item in the array. Array expansion creates a new array containing just the `Name` properties of all books in the array.

If a variable named `myArrayOfBooks` holds your array, use the following code to extract the `Name` properties:

```
var nameArray = myArrayOfBooks.Name
```

The `nameArray` variable contains an array whose length is exactly the same as the length of `myArrayofBooks`. The first item is the value `myArrayofBooks[0].Name`, the second item is the value of `myArrayofBooks[1].Name`, and so on.

For another example, suppose you wanted to get a list of the groups a user belongs to so you can display the display names of each group. Suppose a `User` object contains a `MemberGroups` property that returns a read-only array of groups that the user belongs to. In other words, the Gosu syntax `user.MemberGroups` returns an array of `Group` objects, each one of which has a `DisplayName` property. If you want to get the display names from each group, use the following Gosu code

```
user.MemberGroups*.DisplayName
```

Because `MemberGroups` is an array, Gosu expands the array by the `DisplayName` property on the `Group` component type. The result is an array of the names of all the Groups to which the user belongs. The type is `String[]`.

The result might look like the following:

```
["GroupName1", "GroupName2", "GroupName14", "GroupName22"]
```

The expansion operator works with methods also. Gosu uses the type that the method returns to determine how to expand it:

- if the original object is an array, Gosu creates an expanded array
- if the original method is a list, Gosu creates an expanded list

The following example calls a method on the `String` component of the `List` of `String` objects. It generates the list of initials, in other words the first character in each word.

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

// get the character array [F, G]
var charArray = s*.charAt( 0 )
```

Array expansion is valuable if you need a single one-dimensional array or list through which you can iterate. Also, there are various enhancement methods for manipulating arrays and lists. For details, see "Enhancement Reference for Collections and Related Types" on page 188.

Important notes about the expansion operator:

- The generated array or list itself is always **read-only** from Gosu. You can never assign values to elements within the array, such as setting `nameArray[0]`.
- The expansion operator `*.` works only for array expansion, never standard property accessing.
- When using the `*.` expansion operator, only **component type properties** are accessible.
- When using the `*.` expansion operator, **array properties** are never accessible.

- The expansion operator applies not only to arrays, but to any `Iterable` type and all `Iterator` types and it preserves the type of array/list. For instance, if you apply the *. operator to a `List`, the result is a `List`. Otherwise, the expansion behavior is the same as with arrays.

Array Flattening to Single Dimensional Array

If the property value on the original item returns an array of items, expansion behavior is slightly different. Instead of returning an array of arrays (an array where every item is an array), Gosu returns an array containing all individual elements of all the values in each array.

Some people refer to this approach as *flattening* the array.

To demonstrate this, create the following test Gosu class:

```
package test

class Family {
  var _members : String[] as Members
}
```

Next, paste the following in to the Gosu Tester window

```
uses java.util.Map
uses test.Family

// create objects that each contain a Members property that is an array
var obj1 = new Family() { :Members = {"Peter", "Dave", "Scott"} }
var obj2 = new Family() { :Members = {"Carson", "Gus", "Maureen"} }

// Create a list of objects, each of which has an array property
var familyList : List<Family> = {obj1, obj2}

// List expansion, with FLATTENING of the arrays into a single-dimensional array
var allMembers = familyList*.Members

print(allMembers)
```

This program prints the following single-dimensional array:

```
["Peter", "Dave", "Scott", "Carson", "Gus", "Maureen"]
```

Enhancement Reference for Collections and Related Types

The collection and list classes used frequently in Gosu rely on the Java language's collections and lists classes. However, there are important differences because of built-in *enhancements* to these classes. Gosu *enhancements* are additions to a class or other type that directly add Gosu methods and/or properties to the type, without requiring subclassing to make use of the new features. This is especially useful if you extend Java classes that do not support features, such as adding APIs that use *Gosu blocks*.

Combining Gosu enhancements and Gosu blocks permits concise easy-to-understand Gosu code that manipulates collections. With a single line of code, you can loop across collection items to perform actions on each item, extract information from each item, or sort the collection. For example, Gosu adds the methods `map`, `each`, `sortby`, and other methods to classes.

The following table lists some of the collection enhancements. The letter `T` refers to the type of the collection. The syntax <T> relates to the feature *Gosu generics*, discussed in "Gosu Generics" on page 173. For example, suppose the argument is listed as:

```
conditionBlock(T):Boolean
```

This means the argument is a block. That block must take exactly one argument of the list's type (`T`) and returns a `Boolean`. Similarly, where the letter `Q` occurs, this represents another type. The text at the beginning (in that example, `conditionBlock` is a parameter that is a block and its name describes the block's purpose.

Note: If a type letter wildcard like `T` or `Q` appears more than once in arguments or return result, it must represent the *same type* each time that letter is used.

Collections Enhancement Methods

Gosu contains enhancement methods for Java collection-related types.

Enhancement Methods on Iterable<T>

Iterable objects (objects that implement `Iterable<T>`) have additional methods described in the following table.

Method/Property	Description
`Count`	Returns the number of elements in the `Iterable`
`single()`	If there is only one element in the Iterable, that value is returned. Otherwise an `IllegalStateException` is thrown.
`toCollection()`	If this Iterable is already of type `Collection`, return it. Otherwise, copy all values out of this Iterable into a new Collection.

Enhancement Methods on Collection<T>

Most collection methods are now implemented directly on `Collection` (not `List` or other similar objects as in previous releases). The following table lists the available methods.

Method/Property Name	Description
`allMatch(cond)`	Returns true if all elements in the `Collection` satisfy the condition
`hasMatch(cond)`	Returns true if this `Collection` has any elements in it that match the given block
`asIterable()`	Returns this `Collection<T>` as a pure `Iterable<T>` (in other words, not as a `List<T>`).
`average( selector )`	Returns the average of the numeric values selected from the `Collection<T>`
`countWhere(cond)`	Returns the number of elements in the `Collection` that match the given condition
`HasElements`	Returns `true` if this `Collection` has any elements in it. This is a better method to use than the default collection method `empty()` because `HasElements` interacts better with `null` values. For example, the expression `col.HasElements()` returns a non-true value even if the expression `col` is `null`.
`first()`	Returns first element in the `Collection`, or return `null` if the collection is empty.
`firstWhere(cond)`	Returns first element in the `Collection` that satisfies the condition, or returns `null` if none do.
`flatMap(proj)`	Maps each element of the `Collection` to a `Collection` of values and then flattens them into a single `List`.
`fold()`	Accumulates the values of an `Collection<T>` into a single `T`.
`intersect(iter)`	Returns a `Set<T>` that is the intersection of the two Collection objects.
`last()`	Returns last element in the `Collection` or return `null` if the list is empty.
`lastWhere(cond)`	Returns last element in the `Collection` that matches the given condition, or `null` if no elements match it.
`map(proj)`	Returns a `List` of each element of the `Collection<T>` mapped to a new value.
`max(proj)`	Returns maximum of the selected values from `Collection<T>`
`min(proj)`	Returns minimum of the selected values from `Collection<T>`
`orderBy(proj)`	Returns a new `List<T>` ordered by a block that you provide. Note that this is different than `sortBy()`, which is retained on `List<T>` and which sorts in place. **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.
`orderByDescending(proj)`	Returns a new `List<T>` reverse ordered by the given value. Note that this is different than `sortByDescending()`, which is retained on `List<T>` and which sorts in place. **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.

Method/Property Name	Description
`partition(proj)`	Partitions this `Collection` into a Map of keys to a list of elements in this `Collection`.
`partitionUniquely(proj)`	Partitions this `Collection` into a Map of keys to elements in this `Collection`. Throws an `IllegalStateException` if more than one element maps to the same key.
`reduce(init, reducer)`	Accumulates the values of a `Collection<T>` into a single V given an initial seed value.
`reverse()`	Reverses the collection as a `List`.
`singleWhere(cond)`	If there is only one element in the `Collection` that matches the given condition, it is returned. Otherwise an `IllegalStateException` is thrown
`sum(proj)`	Returns the sum of the numeric values selected from the `Collection<T>`
`thenBy(proj)`	Additionally orders a `List` that has already been ordered by `orderBy`. **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.
`thenByDescending(proj)`	Additionally reverse orders a `List` that has already been ordered by `orderBy`. **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.
`toList()`	If this `Collection` is already a list, simply return it. Otherwise create a new `List` and copy this `Collection` to it.
`toTypedArray()`	Converts this `Collection<T>` into an array `T[]`.
`union(col)`	Returns a new `Set<T>` that is the union of the two Collections
`where(cond)`	Returns all elements in this Iterable that satisfy the given condition
`whereTypeIs(Type)`	Returns a new `List<T>` of all elements that are of the given type
`disjunction()`	Returns a new `Set<T>` that is the set disjunction of this collection and the other collection
`each()`	iterates each element of the `Collection`
`eachWithIndex()`	Iterates each element of the `Collection` with an index
`join`	joins all elements together as a string with a delimiter
`minBy()`	Returns the minimum T of the `Collection` based on the projection to a `Comparable` object
`maxBy()`	Returns the maximum T of the `Collection` based on the projection to a `Comparable` object
`removeWhere()`	Removes all elements that satisfy the given criteria
`retainWhere()`	Removes all elements that do not satisfy the given criteria. This method returns no value, so it cannot be chained in series. This is to make clear that the mutation is happening in place, rather than a new collection created with offending elements removed.
`subtract()`	Returns a new `Set<T>` that is the set subtraction of the other collection from this collection
`toSet()`	Converts the `Collection` to a `Set`

Methods on List<T>

The following table lists the available methods on `List<T>`.

Method/Property Name	Description
`reverse()`	Reverses the Iterable.
`copy()`	Creates a copy of the list
`freeze()`	Returns a new unmodifiable version of the list
`shuffle()`	Shuffles the list in place
`sort()`	Sorts the list in place

Method/Property Name	Description
`sortBy()`	Sorts the list in place in ascending order **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.
`sortByDescending()`	Sorts the list in place in descending order. **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.

Methods on Set<T>

The following table lists the available methods on `Set<T>`.

Method/Property Name	Description
`copy()`	Creates a copy of the set
`powerSet()`	Returns the power set of the set
`freeze()`	Returns a new unmodifiable version of the set

The following subsections describe the most common uses of these collection enhancement methods.

Finding Data in Collections

You probably frequently need to find an items in a list based on certain criteria. Use the `firstWhere` or `where` methods in such cases. These functions can be very processor intensive, so be careful how you use them. Consider whether other approaches may be better, testing your code as appropriate.

The `where` method takes a block that returns `true` or `false` and return all elements for which the block returns `true`. The following demonstrates this method:

```
var strs = new ArrayList<String>(){"a", "ab", "abc"}
var longerStrings = strs.where( \ str -> str.length >= 2 )
```

The value of `longerStrings` is `{"ab", "abc"}`. The expression `str.length >= 2` is `true` for both of them.

The `firstWhere` method takes a block that returns `true` or `false` and return the first elements for which the block returns `true`. The following example demonstrates how to find the first item that matches the criteria:

```
var strs = new ArrayList<String>(){"a", "ab", "abc"}
var firstLongerStr = strs.firstWhere( \ str -> str.length >= 2 )
```

The value of `firstLongerStr` is `"ab"`, since `"ab"` is the first element in the list for which `str.length >= 2` evaluates as `true`.

If `firstWhere` finds no matching items, it returns `null`.

Similarly, there is a `lastWhere` method that finds the last item that matche the condition, and returns `null` if none are found.

Sorting Collections

Suppose you had an array list of strings:

```
var myStrings = new ArrayList<String>(){"a", "abcd", "ab", "abc"}
```

You can easily resort the list by the length of the `String` values using blocks. Create a block that takes a `String` and returns the sort key, which in this case is the number of characters of the parameter. Let the `List.sortBy(...)` method handle the rest of the details of the sorting and return the new sorted array as the result.

```
var resortedStrings = myStrings.sortBy( \ str -> str.Length )
```

If you want to print the contents, you could print them with:

```
resortedStrings.each( \ str  -> print( str )  )
```

...which would produce the output:

```
a
ab
abc
abcd
```

Similarly, you can use the `sortByDescending` function, which is the same except that it sorts in the opposite order.

For both of these methods, the block must return a comparable value. Comparable values include `Integer`, a `String`, or any other values that can be compared with the “>” or “<” (greater than or less than) operators.

In some cases, comparison among your list objects might be less straightforward. You might require more complex Gosu code to compare two items in the list. In such cases, use the more general sort method simply called `sort`. The `sort` method takes a block that takes two elements and returns `true` if the first element comes before the second, or otherwise returns `false`. The earlier sorting example could be written as:

```
var strs = new ArrayList<String>(){"a", "abc", "ab"}
var sortedStrs = strs.sort( \ str1, str2 -> str1.length < str2.length )
```

Although this method is powerful, in most cases code is more concise and easier to understand if you use the `sortBy` or `sortByDescending` methods instead of the `sort` method.

> **Note:** The collection enhancement methods for sorting and ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. Because of this, these methods do not sort `String` values in a locale-sensitive way.

Mapping Data in Collections

Suppose you want Gosu code to take an array list of strings and find the number of characters in each string. Use the list method `map` to create a *new* list where the expression transforms each value and makes the result an element in a new list.

For example:

```
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}
var lengthsOnly = myStrings.map( \ str -> str.length )
```

The value of `lengthsOnly` at the end of this code is an array with elements: 1, 1, 2, 2, 3, 4.

In this example, the `map` method takes a block that is a simple function taking one `String` and returning its length. However, notice that it did not explicitly set the type of the block’s argument called `myStrings`. However, this is not an untyped argument, at compile time it is statically typed as a `String` argument. This is implicit because the array list is specified as a list of `String` using the generics syntax `ArrayList<String>`.

Some Gosu collection-related code has concise syntax because collection methods use Gosu *generics*. Generics allow methods such as `map` to naturally define the relationship of types in arguments, return values, and the type of objects in the collection. In this case, the array list is an array list of strings. The `map` method takes a block that **must** have exactly one argument and it **must** be a `String`. Gosu knows the block **must** take a `String` argument so the type can be omitted. Gosu can simply infer the argument type to allow flexible concise code with all the safety of statically-typed code.

The type of the `lengthsOnly` variable also uses type inference and is statically typed. Because the block returns an `int`, the result type of the function must be an `int`. Because of this, `lengthsOnly` is statically typed at compile time to an array of integers even though the type name is not explicit in the code. Specifying the type is optional, and it is good Gosu coding style to use type inference for simple cases like this.

Iterating Across Collections

Now suppose you also want to print each number in the list. You could take advantage of the list method `each`, which can be used in place of a traditional Gosu loop using the `for` keyword:

```
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}
myStrings.map( \ str -> str.length ).each( \ len  -> print( len ) )
```

As you can see, this is a simple and powerful way to do some types of repeated actions with collections. This conciseness can be good or bad, depending on the context of the code. In some cases, it might be better to assign the return value of `map` to a variable and call the `each` method on it. This is especially true if you still need the array of lengths even after printing them. For example:

```
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}
var strLengths = myStrings.map( \ str -> str.length )
strLengths.each( \ len  -> print( len )  )

// maybe use strLengths again in some way here...
```

This is equivalent and some people may find it easier to read.

Partitioning Collections

Blocks are also useful with the `partition` method. This method takes a list and creates a new `java.util.Map` of key/value pairs. The block takes an item from the original list as an argument and returns a value. To perform this task for all input list items, the map keys are results from the block with the input list. Each key points to the input list items that produced that value.

For example, suppose you want take a `String` list and partition it into a `Map` containing the lengths of each `String` value. Suppose the set of input values were the following:

```
var myStrings = new ArrayList<String>(){"a", "b", "bb", "ab", "abc", "abcd"}
```

Each key points to a list of all input `String` values with that length. You could use this one line of Gosu code:

```
var lengthsToStringsMap = myStrings.partition( \ str:String -> str.length )
```

The variable `lengthsToStringsMap` contains a `Map` with four keys:

```
Map { 1 → ["a", "b"], 2 → ["bb", "ab"], 3 → ["abc"], 4 → ["abcd"] }
```

In other words:

- key `1` points to a list of two values, `"a"` and `"b"`
- key `2` points to a list of two values `"bb"` and `"ab"`
- key `3` points to a list with a single value, `"abc"`
- key `4` points to a list with a single value, `"abcd"`

As you can tell from this example, you can make concise and easy-to-read Gosu code with powerful results. Also, note the resulting `Map` is statically typed using type inference.

You can improve your performance if you are sure the output of your block for each list element is always unique. The indirection of having each value wrapped within a list using the `partition` method is unnecessary because there is always a single item in every list. For faster performance in the case in which you know block return results are unique, use the `partitionUniquely` method.

For example:

```
var myStrings = new ArrayList<String>(){"bb", "a", "abcd", "abc"}
var lengthsToStringsMap = myStrings.partitionUniquely( \ str:String -> str.length )
```

The result `Map` has values that are single items not lists:

```
Map { 1 → "a", 2 → "bb", 3 → "abc", 4 → "abcd" }
```

In a real-world situation, you might use code like:

```
//Use a finder to find get a list of claims
var claims = find claim in Claim where ...

//partition the list
var claimsById = claims.partitionUniquely( \ claim -> claim.publicID )
```

The value of `claimsById` is a `Map` of claim `publicID` values to the claims they represent.

If more than one element of the list has the same calculated value for the attribute, the method throws a runtime exception.

Converting Lists, Arrays, and Sets

Use the collection enhancements to convert lists, arrays, and sets as necessary to other types:

- You can convert a `List` or an `Array` to a set by calling `list.toSet()` or `array.toSet()`.
- You can convert a `Set` or an `Array` to a list by calling `set.toList()` or `array.toList()`.
- You can join all of the elements in an `Array` or `List` together with a delimiter by the `join` method, such as:

```
// join all the items in the array together separated by commas
joinedString = array.join(",")
```

Flat Mapping a Series of Collections or Arrays

Use the `flatMap` method to create a single `List` from a series of collections or arrays of objects associated with properties on elements on an outer collection. You provide a block that takes an element of a collection and returns an array or collection, such as an array or collection stored a property of the outer collection. The `flatMap` method concatenates all the items in the returned arrays or collections into a single `List`.

For example, suppose the following structure of your data:

- a claim object has an `Exposures` property that contains an array of exposure objects
- an exposure has a `Notes` property that contains a list of `Note` objects.

First, write a simple block that extracts the note objects from the exposure object:

```
\ e -> e.Notes
```

Next, pass this block to the `flatMap` method to generate a single list of all notes on the claim:

```
var allNotes = myClaim.Exposures.flatMap( \ e -> e.Notes )
```

This generates a single list of notes (on instance of `List<Note>` in generics notation) containing all the notes on all the exposures on the claim.

This method is similar to the Gosu feature called *array expansion* (see “Array Expansion” on page 74). However, it is available on all collections as well as arrays, and `flatMap` method generate different extracted arrays dynamically using a Gosu block that you provide. Your block can perform any arbitrary and potentially-complex calculation during the flat mapping process.

Sizes and Length of Collections and Strings are Equivalent

Gosu adds enhancements for the `Collection` and `String` classes to support both the `length` and `size` properties, so you can use the terms interchangeably with no errors. For collections and strings, `length` and `size` mean the same thing in Gosu.

chapter 18

Gosu and XML

XML files describe complex structured data in a text-based format with strict syntax for easy data interchange. For more information on the Extensible Markup Language, refer to the World Wide Web Consortium web page `http://www.w3.org/XML`. Gosu can read or write any XML document. If you have an associated XSD to define the document structure, Gosu parses the XML using the schema. This produces a statically-typed tree of XML elements with structured data. Also, during parsing, Gosu can validate the XML against the schema. You can also 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 (Gosu properties on each element) or intelligent static typing.

This topic includes:

Manipulating XML Overview

To manipulate XML in Gosu, Gosu creates an in-memory representation of a graph of XML elements. The main Gosu class to handle an XML element is the class called `XmlElement`. Instead of manipulating XML by modifying text data in an XML file, your Gosu code can simply manipulate `XmlElement` objects. You can read in XML data from a file or other sources and parse it into a graph of XML elements. You can export a graph of XML elements as standard XML, for example as an array of bytes containing XML data.

Gosu can manipulate structured XML documents in two ways:

- **Untyped nodes.** Any XML can be easily created, manipulated, or searched as a tree of untyped nodes. For those familiar with Document Object Model (DOM), this approach is similar to manipulating DOM untyped nodes. From Gosu, attribute and node values are treated as strings.
- **Strongly typed nodes using an XSD.** If the XML has an XML Schema Definition file (an *XSD file*), you can create, manipulate, or search data with statically-typed nodes that correspond to legal attributes and child elements. If you can provide an XSD file, the XSD approach is much safer. It dramatically reduces errors due to incorrect types or incorrect structure.

Introduction to XmlElement

The main class that represents an XML element is the class `XmlElement`.

An `XmlElement` object consists of the following items (and only the following items):

- **The element name (as a QName).** The element's name is not simply a `String` value. It is a fully-qualified name (a *QName*). A QName represents a more advanced definition of a name than a simple `String` value. Gosu uses the standard Java way to specify a QName: the class `javax.xml.namespace.QName.` A QName object contains the following components:
  - A `String` value that represents the local part (also called the `localPart`)
  - A `String` value that represents the namespace URI that the local part of the name is defined within. For example, a namespace might have the value: `http://www.w3.org/2001/XMLSchema-instance`
  - A *suggested* prefix name if Gosu later serializes this element. (This prefix is not guaranteed upon serialization, since there may be conflicts.)

  For example, you might see in an XML file an element name with the syntax of two parts separated by a colon, such as `veh:root`. The `root` part of the name is the local part. The prefix `veh` in this example indicates that the XML document (earlier in the file) a declared namespace and a shortcut name (the prefix `veh`) to represent the full URI.

  For example, consider the following XML document:

```
<?xml version="1.0"?>
<veh:root xmlns:veh="http://mycompany.com/schema/vehiclexsd">
  <veh:childelement/>
</veh:root>
```

  The following things are true about this XML document:

  - The root element of the document has the name `root` within the namespace `http://mycompany.com/schema/vehiclexsd`.
  - The text `xmlns:veh` 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, whose name is `childelement` within the namespace `http://mycompany.com/schema/vehiclexsd`. However, this XML document specifies the namespace not with the full URI but with the shortcut prefix `veh` followed by the colon (and then followed by the local part).

  There are three constructors for QName:

- QName constructor specifying the namespace URI, local part, and suggested prefix.

```
QName(String namespaceURI, String localPart, String prefix)
```

- QName constructor specifying the namespace URI and local part (suggested prefix is implicitly empty).

```
QName(String namespaceURI, String localPart)
```

- QName constructor specifying the local part only (the namespace and URL are implicitly empty)

```
QName(String localPart)
```

You can set the namespace in the QName to the empty namespace, which technically is the constant `javax.xml.XMLConstants.NULL_NS_URI`. The recommended approach for creating QName objects in the empty namespace is to use the QName constructor that does **not** take a namespace argument.

To create multiple `QName` objects easily in the same namespace, you can use the optional utility class called `XmlNamespace`. For details, see "Creating Many QNames in the Same Namespace" on page 203.

When constructing an `XmlElement`, the name is strictly required and must be non-empty.

Note: QNames are used for other purposes in Gosu XML APIs. 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 QNames.

- **A backing type instance.** Each element contains a reference to a Gosu type that represents this specific element. To get the backing type instance, get the `TypeInstance` property from the element. For XML elements that Gosu created based on an XSD, Gosu sets this backing type information automatically so it can be used in a typesafe manner.

  When constructing 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. See "Getting Data From an XML Element" on page 211 for related information

  The type instance of an XML element is responsible for most of the element's behavior but does not contain the element's name. You can sometimes ignore the division of labor between an `XmlElement` and its backing type instance. If you are using an XSD, this distinction is useful and sometimes critical. For more information, see "Getting Data From an XML Element" on page 211.

- **The nillness of the element**. XML has a concept of whether an element is `nil`. This is not exactly same as being `null`. An element can be `nil` (and must have no child elements) but *still have attributes*. Additionally, an XSD can define whether an element is *nillable*, which means that element is allowed to be `nil`. For more information, see "Access the Nillness of an Element" on page 217.

To summarize, the `XmlElement` instance contains the properties shown in the following table.

XmlElement property	Type	Description
`QName`	`QName`	A read-only property that returns the element's QName.
`Namespace`	`XmlNamespace`	Returns an `XmlNamespace` object that represents the element's namespace
`TypeInstance`	`gw.xsd.w3c.xmlschema.types.complex.AnyType` or any subclass of that class	Returns the element's backing type instance
`Nillness`	`boolean`	Specifies whether this element is `nil`, which is an XML concept that is not the same as being `null`. See "Access the Nillness of an Element" on page 217

IMPORTANT If you are accessing these properties on an XSD-based element, you must use a dollar sign prefix for the property name. See "Dollar Sign Prefix For Some Properties When Using XSD Types" on page 199

To create a basic `XmlElement`, simply pass the element name to the constructor as either a `QName` or a `String`. The constructor on `XmlElement` that takes a `String` is a convenience method. The `String` constructor is equivalent passing a new QName with that String as the one-argument constructor to QName. In other words, the namespace and prefix in the QName are `null` if you use the `String` constructor on `XmlElement`.

The following code creates an in-memory Gosu object that represents an XML element `<Root>` in the empty namespace:

```
var el = new XmlElement( "Root" )
```

In this case, the `el.TypeInstance` property returns an instance of the default type `gw.xsd.w3c.xmlschema.types.complex.AnyType`. If you instantiate a type instance, typically you would use more specific subclass of `AnyType`, either an XSD-based type or a simple type.

For a more complex example, the following Gosu code creates a new `XmlElement` without an XSD, and adds a child element:

```
uses gw.xml.XmlElement
uses javax.xml.namespace.QName

var e = new XmlElement(new QName("http://mycompany.com/schema/vehiclexsd", "root", "veh"))
var e2 = new XmlElement(new QName("http://mycompany.com/schema/vehiclexsd", "childelement", "veh"))

e.addChild(e2)
e.print()
```

This prints the following:

```
<?xml version="1.0"?>
<veh:root xmlns:veh="http://mycompany.com/schema/vehiclexsd">
  <veh:childelement/>
</veh:root>
```

This output is the QName example from earlier in this section.

For more information about adding child elements, see "Getting Data From an XML Element" on page 211.

What Does an Element Contain Inside It?

Gosu exposes properties and methods on the XML type instances to access or manipulate child elements or text contents. It is important to note that XML elements effectively could contain two basic types of content:

- child elements
- a simple value (which can represent simple types such as numbers or dates)

Technically, the Gosu object that represent the element does not directly contain the child elements or the text content. It is the *backing type instance* for each element which contains the text content. However, in practice this distinction is not typically necessary to remember.

An element can contain either child elements or simple values, but never both at the same time. This distinction is important particularly for XSD-based types. Gosu handles properties on elements differently depending on whether it contains a simple value or is a type that can contain child elements.

IMPORTANT Element contain child elements or simple values, but never both at the same time.

Dollar Sign Prefix For Some Properties When Using XSD Types

For the some properties that the documentation mentions, Gosu provides access directly from the XML element even though the actual implementation internally is on the backing type instance.

If an element is not an XSD-based element, simply access the properties directly, such as `element.Children`.

However, if you are using an XSD type, you must prefix the dollar sign ($) before any property name. This convention prevents ambiguity with properties defined on the XSD type or on the type instance that backs that type. For example, suppose the XSD could define a an element's child element literally named `Children`. There would unfortunately be two similar properties with the same name. Gosu prevents ambiguity by requiring the special properties to have a dollar sign prefix if and only if the element is XSD-based:

- To access the children of an XSD-based element, use the syntax `element.$Children`.
- To access the a custom child element named `Children` as defined by the XSD, use the syntax `element.Children`. This is a non-recommended name due to the ambiguity, but Gosu has no problem with it. You may not have control over the XSD format that you are using, so Gosu must disambiguate them.

Notes about this convention:

- This convention only applies to properties defined on XSD-based types.
- It does not apply to methods.
- It does not apply to non-XSD-based XML elements.

For example, suppose you use the root class `XmlElement` directly with no XSD to manipulate an untyped graph of XML nodes. In that case, you can omit the dollar sign because the property names are not ambiguous. There are no XSD types so there is no overlap in namespace.

This affects the following type instance property names that appear on an XML element, listed with their dollar sign prefix:

- `$Attributes`
- `$Class`
- `$Children`
- `$Namespace`
- `$NamespaceContext`
- `$Comment`
- `$QName`
- `$Text`
- `$TypeInstance`
- `$SimpleValue`
- `$Value` - only for elements with an XSD-defined simple content

- `$Nil` - only for XSD-defined nillable elements. See "Access the Nillness of an Element" on page 217.

IMPORTANT It is important to understand that some special property names on an XML element include a dollar sign prefix if and only if the XML element is an XSD type. If you create an `XmlElement` element directly (not a subclass), it is not an XSD type. It is an untyped node that uses the default type instance (an instance of the type `AnyType`). In such cases, there is no dollar sign prefix because there is no ambiguity between properties that are really part of the type instance, rather than on the XSD type.

Exporting XML Data

The `XmlElement` class includes the following methods and properties that export XML data. All of these methods have alternate method signatures that takes a serialization options object of type `XmlSerializationOptions`. See later in this topic for details of this customization.

Each XML element provides the following methods that serialize the XML element:

- `bytes` method – This method returns an array of bytes (the type `byte[]`) containing the UTF-8-encoded bytes in the XML. Generally speaking, this is the best approach for serializing the XML.
  ```
  var ba = element.bytes()
  ```
  Compare and contrast with the `asUTFString` method, mentioned in the next bullet point.

  If your code sends XML with a transport that only understands character (not byte) data, always base-64 encode the bytes to compactly and safely encode binary data. To do this, use the Gosu syntax:
  ```
  var base64String = gw.util.Base64Util.encode(element.bytes())
  ```
  To reverse the process in Gosu, use the code:
  ```
  var bytes = gw.util.Base64Util.decode(base64String)
  ```
- `print` method – serializes the element to the standard output stream (`System.out`). For example:
  ```
  element.print()
  ```
- `writeTo` method – Writes to an output stream (`java.io.OutputStream`). This method does **not** close the stream afterward.
- `asUTFString` method – serializes the element to a `String` object in UTF-8. For example:
  ```
  var s = element.asUTFString()
  ```
  This method outputs the node as a `String` value containing XML with a header suitable for later export to UTF-8 or UTF-16 encoding. The generated XML header does **not** specify the encoding. In the absence of a specified encoding, all XML parsers must autodetect the encoding (UTF-8 or UTF-16). The existence of a *byte order mark* at the beginning of the document tells the parser what encoding to use. For more details of the XML byte order mark, refer to: `http://www.w3.org/TR/REC-xml/#sec-guessing`

  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. In general, use the `bytes` method to get an array of bytes. If your code sends or stores XML with a transport that only understands character data (not byte data), always `Base64` encode the array of bytes. See the example earlier in this section for the `bytes` method.

  For more information about UTF-8, refer to:
  ```
  http://tools.ietf.org/html/rfc3629
  ```

For all serializations, be sure to test your code with non-English characters. In other words, be sure to test with characters with high Unicode code points.

WARNING Always test your XML serialization and integration code with non-English characters.

For all of these methods, you can customize serialization by optionally passing an `XmlSerializationOptions` instance as another parameter at the end of the parameter list. The serialization options object contains the following properties:

- `Comments` - `Boolean`. If true, exports each element's comments. The default is `true`.
- `Pretty` - `Boolean`. If `true`, Gosu attempts to improve visual layout of the XML with indenting and line separators. The default is `true`. If you set this to `false`, then Gosu ignores the values of the `Indent` and `LineSeparator` properties.
- `Indent` - `String`. Specifies the String to export for each level of hierarchy. The default is two spaces.
- `LineSeparator` - `String`. Sets the line separator. The default is newline (ASCII 10).
- `Sort` - `Boolean`. If true, ensures that the order of children elements of each element match the XSD. The default is `true`. 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.
- `XmlDeclaration` - Returns whether to write the XML declaration at the top of the file.

For example, the following example creates an element, then adds an element comment. Next, it demonstrates printing the element with the default settings (with comments) and how to customize the output to omit comments.

```
uses gw.xml.XmlSerializationOptions

// create an element
var a =  new com.guidewire.pl.docexamples.gosu.xml.simpleelement.MyElement()

// add a comment
a.$Comment = "Hello I am a comment"

print("print element with default settings...")
a.print()

print("print element with no comments...")
a.print(new XmlSerializationOptions()  { :Comments = false})
```

Note that all serialization APIs generate XML data for the entire XML hierarchy with that element at the root.

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.

For each source of data, there is an optional method variant that modifies the way Gosu parses the XML. Gosu encapsulates these options in an instance of the type `XmlParseOptions`. The `XmlParseOptions` specifies additional schemas that resolve schema components for the input instance XML document. Typical code does not need this. Use this if your XML data contains references to schema components that are neither directly nor indirectly imported by the schema of the context type. For more information, refer to later in this topic.

For example, the following simple example parses XML contained in a `String` into an `XmlElement` object, and then prints the parsed XML data:

```
var a = XmlElement.parse( "<Test123/>" )
a.print()
```

If you are using an XSD, call the `parse` method directly on your XSD-based node, which is a subclass of `XmlElement`. For example:

```
var a = com.guidewire.pl.docexamples.gosu.xml.demoattributes.Element1.parse(xmlDataString)
```

The following table lists the parsing methods (for details of `XmlParseOptions`, see “Referencing Additional Schemas During Parsing” on page 202):

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. For more information, see “Referencing Additional Schemas During Parsing” on page 202.

If the XML document fails any of these tests, Gosu throws an exception.

Referencing Additional Schemas During Parsing

In some advanced parsing situations, you might need to reference additional schemas other than your main schema during parsing.

To specify additional schemas, set the `XmlParseOptions.AdditionalSchemas` to a specific `SchemaAccess` object. This `SchemaAccess` object represents the XSD. To access it from an XSD, use the syntax:

```
package_for_the_schema.util.SchemaAccess
```

To see how and why you would use this, suppose you have the following two schemas:

The XSD `ImportXSD1.xsd`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="urn:ImportXSD1"
    xmlns:ImportXSD1="urn:ImportXSD1">
    <xsd:element name="ElementFromSchema1" type="ImportXSD1:TypeFromSchema1"/>
    <xsd:complexType name="TypeFromSchema1"/>
</xsd:schema>
```

The XSD `ImportXSD2.xsd`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="urn:ImportXSD2"
  xmlns:ImportXSD1="urn:ImportXSD1" xmlns:ImportXSD2="urn:ImportXSD2" elementFormDefault="qualified">
    <xsd:import schemaLocation="ImportXSD1.xsd" namespace="urn:ImportXSD1"/>
    <xsd:complexType name="TypeFromSchema2">
        <xsd:complexContent><!-- the TypeFromSchema2 type extends the TypeFromSchema1 type! -->
            <xsd:extension base="ImportXSD1:TypeFromSchema1"/>
        </xsd:complexContent>
    </xsd:complexType>
</xsd:schema>
```

Notice that the `ImportXSD2` XSD extends a type that the `ImportXSD1` defines. This is analogous to saying the `ImportXSD2` type called `TypeFromSchema2` is like a subclass of the `ImportXSD1` type called `TypeFromSchema1`.

The following code fails (throws exceptions) because the `ImportXSD1` references the schema type `ImportXSD2:TypeFromSchema2` and Gosu cannot find it anywhere in the current schema.

```
var schema2 = com.guidewire.pl.docexamples.gosu.xml.importxsd2.util.SchemaAccess

var xsdtext =  "<ElementFromSchema1 xmlns=\"urn:ImportXSD1\" xmlns:ImportXSD2=\"urn:ImportXSD2\"" +
 " xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" xsi:type=\"ImportXSD2:TypeFromSchema2\"/>"

// parse an element defined in the the first schema, but pass an extension to that
// that type that the second schema defines. THIS FAILS without using the AdditionalSchemas feature
var element = ElementFromSchema1.parse(xsdtext)
```

The main problem is that the `ImportXSD1` XSD type does not directly know about the existence of the schema called `ImportXSD2` even though it extends one of its types.

To make it work, set the `AdditionalSchemas` property of the `XmlParseOptions` object to a list containing one or more `SchemaAccess` objects. In other words, the following XML parsing code succeeds:

```
var schema2 = com.guidewire.pl.docexamples.gosu.xml.importxsd2.util.SchemaAccess
var options = new gw.xml.XmlParseOptions() { :AdditionalSchemas = { schema2 } }

var xsdtext =  "<ElementFromSchema1 xmlns=\"urn:ImportXSD1\" xmlns:ImportXSD2=\"urn:ImportXSD2\"" +
 " xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" xsi:type=\"ImportXSD2:TypeFromSchema2\"/>"

// parse an element defined in the the first schema, but pass an extension to that
// that type that the second schema defines. To do this, it requires using the XmlParseOptions
var element = ElementFromSchema1.parse(xsdtext, options)
```

Creating Many QNames in the Same Namespace

The name of each element has the type QName, which is an object of type `javax.xml.namespace.QName`. For more information, see “Introduction to XmlElement” on page 196.

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"`. However, in some cases 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. The direct way to do this is to store the namespace URI into 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 this:

- Create an `XmlNamespace` directly and call its `qualify` method and pass the local part `String`. For example:

```
uses gw.xml.XmlNamespace
var ns = new XmlNamespace("namespaceURI","prefix")
var e = new XmlElement(ns.qualify("localPartName"))
```

- Reuse the namespace of an already-created XML element. To get the namespace from an XML element instance, get its `NameSpace` property. Then, simply call the qualify method and pass the local part `String`:

```
// create a new XML element
var xml = new XmlElement( new QName( "namespaceURI", "localPart", "prefix" ) )

// shorthand for reusing the namespaceURI and prefix from the previously-created element
var xml2 = new XmlElement( xml.Namespace.qualify( "localPart2" ) )
```

XSD-based Properties and Types

The most powerful way to use XML in Gosu is to use an XSD that describes in a strict way what is valid in your XML. If you can use or generate an XSD for your data, it is strongly recommended to use an XSD.

You load an XSD into the Gosu type system by putting it in the Gosu class hierarchy. Gosu creates new types in the type system for element declarations in the XSD. Where appropriate, Gosu creates properties on these types based on the names and structure within the XSD. By using an XSD and the generated types and properties, your XML-related code is significantly easier to read and understand. For example, you can use natural Gosu syntax to access child elements by their name such as `element.ChildName` for a child named `ChildName`.

If you cannot use an XSD, you can use the basic properties and methods of `XmlElement` like `element.Children` and `element.getChild("ChildName")`. However, writing XML-related code without XSD types tends to be harder to understand getting and setting values and elements, and much less typesafe.

Important Concepts in XSD Properties and Types

There are some important distinctions to make in terminology when understanding how Gosu creates types from XSDs. In the following table, note how every definition in the XSD has a corresponding instance in an XML document (although in some cases might be optional).

Definitions (in the XSD)	Instances (in an XML document)
a schema (an XSD)	XML document
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 (for example, a top-level complex type)
  - *it embeds* a type definition inside the element definition (for example, an embedded simple type)
  - it includes no type, which implicitly refers to the built-in complex type `<xsd:anyType>`

In an XSD, various definitions cause Gosu to create new types:

- an element definition causes Gosu to create a type that describes the element
- a type definition causes Gosu to create a type that describes the type (for example, a new complex type)
- an attribute definition causes Gosu to create a type that describes the attribute

For example, suppose an XSD declares a new top-level simple type that represents a phone number. Suppose there are 3 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:

- 1 type that represents the phone number **simple type**
- 3 types that represent the individual **element definitions that reference the phone number**

From Gosu, when you are creating objects or setting properties on elements, it is important to know which type you want to use. In some cases, you might be able to do what you want in more than one way, although one way might be easier to read. See "XSD Generated Type Examples" on page 208 for examples that illustrate this point further.

Also remember that if you have a reference to the element, you can always reference the backing type. For example, for an element, you can reference the backing type instance using the `$TypeInstance` property. See "XSD Generated Type Examples" on page 208 for examples of this.

Reference of XSD Properties and Types

The following table lists the types and properties that Gosu creates from an XSD. For this topic, *`schema`* represents the fully-qualified path to the schema, *`elementName`* represents an element name, and *`parentName`* and *`childName`* represent names of parent and child elements.

The rightmost column indicates (for properties only) whether the property becomes a list property if it can appear more than once. If it says "Yes", the property has type `java.util.List` parameterized on what type it is when it is singular. For example, suppose a child element is declared in the XSD with the type `xsd:int`:

- If its `maxOccurs` is 1, the property's type is `Integer`
- If its `maxOccurs` is *greater than 1*, the property's type is `List<Integer>`, which means a list of integers

There are other circumstances in which a property becomes a list. For example, suppose there is a XSD choice (`<xsd:choice>`) in an XSD that has `maxOccurs` attribute value greater than 1. Any child elements become list properties. For example, if the choice defines child elements with names `"elementA"` and `"elementB"`, Gosu creates properties called `ElementA` and `ElementB`, both declared as lists. Be aware that Gosu exposes shortcuts for inserting items, see "Automatic Insertion into Lists" on page 209.

Notes about generated types containing the text `anonymous` in the fully qualified type name:

- Although the package includes the word `anonymous`, this does not imply that these elements have no defined names. The important quality that distinguishes these types is that the object is defined at a **lower level** than the top level of the schema. By analogy, this is similar to how Gosu and Java define *inner classes* within the namespace of another class.
- There are several rows that contain a reference to the path from root as the placeholder text *`PathFromRoot`*. The path from root is a generated name that embeds the path from the root of the XSD, with names separated

by underscore characters. The intermediate layers may be element names or group names. See each row for examples.

For each occurrence of...	Declared in the schema at this location...	There is a new...	With syntax...
element definition	top level	type	*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*.
	lower than 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*.
complex type definition	top level	type	*schema*.types.complex.*TypeName*
	lower than 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.
simple type definition	top level	type	*schema*.types.simple.*TypeName*
	lower than top level	type	*schema*.anonymous.types.simple.*PathFromRoot* For example, suppose a top level element A contains element B, which contains an embedded simple type. The path from root is A_B. Note that simple types defined at a level lower than the top level never have names.
attribute definition	top level	type	*schema*.attributes.*AttributeName*
	lower than top level	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*.
	within an element	property	*element*.*AttributeName* Unlike most other generated properties on XSD types, an attribute property **never** transform into a list property.
For every child element with either (1) simple type or (2) complex type and a simple content			
It is a common pattern to convert a simpleType at a later time to simpleContent simply to add **attributes** to an element with a simple type. To support this common pattern, Gosu creates two properties ChildName and ChildName_elem for every child element with either a simply type or both a complex type and simple content. The one with the _elem suffix 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, your XML code requires no changes simply because of this change.			
child element with either: • simple type • complex type and a simple content	*anywhere*	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:** This property transforms into a list type if it can appear more than once. See discussion of list properties immediately before this table.

For each occurrence of...	Declared in the schema at this location...	There is a new...	With syntax...
the *value* of a child element with either: • simple type • complex type and a simple content	*anywhere*	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:** This property transforms into a list type if it can appear more than once. See discussion of list properties immediately before this table.
For every child element with complex type and no simple content			
child element with complex type and no simple content	*anywhere*	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:** This property transforms into a list type if it can appear more than once. See discussion of list properties immediately before this table.
For each schema			
schema definition	*n/a*	schema access object	`schema.util.SchemaAccess` It is a special utility object for providing access to the original schema that produced this type hierarchy. Think of this as a way of representing this schema. This is important if you need one schema to include another schema (see "Referencing Additional Schemas During Parsing" on page 202).

Normalization of Gosu Generated XSD-based Names

In cases where Gosu creates type names and element names, Gosu performs slight normalization of the names:

- One prominent aspect of normalization is capitalization to conform to Gosu naming standards for packages, properties, and types. For example, Gosu packages become all lowercase. Types must start with initial capitals. Properties must start with initial capitals.
- If the type or property names contains invalid characters for Gosu for that context, Gosu changes them. For example, hyphens are disallowed and removed.
  - If Gosu finds an invalid character and the following character is lowercase, Gosu removes the invalid character and uppercases the following letter.
  - If Gosu finds an invalid character and the following character is uppercase, Gosu converts the invalid character to an underscore and does not change the following character.
  - If the first character is invalid as a first character but otherwise valid (for example, a numeric digit), Gosu simply prepends an underscore. If it is entirely invalid within a name in that context (such as hyphen), Gosu removes the character. In the unusual case in which after removing all start characters, no characters remain, Gosu simply renames that item a simple underscore.
- If there are duplicates, Gosu appends numbers to some of them. For example, `MyProp`, `MyProp2`, `MyProp3`, and so on.

XSD Generated Type Examples

XSD Generated Type Examples 1

Let us try these with actual examples. Suppose you have the following XSD in the package `examples.pl.gosu.xml`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Element1">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="Child1"/> <!-- 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 // this has type schema.anonymous.elements.Element1_Child1
var child2 = xml.Child2 // this has type java.lang.Integer
xml.Child2 = 5 // set the property with a simple type
var child2Elem = xml.Child2_elem // this is a schema.anonymous.elements.Element1_Child2
```

Note the following:

- The `Child1` property is of type `schema.anonymous.elements.Element1_Child1`, which is a subclass of `XmlElement`.
- The `Child2` property is of type `java.lang.Integer`. When a child element has a simple type, its natural property name gets the object's **value** (rather than the child element **object**). If you wish to access the element object (the `XmlElement` instance) for that child, instead use the property with the `_elem` suffix. In this case, for the child named `Child2`, you use the element.`Child2_elem` property, which is of type `schema.anonymous.elements.Element1_Child2`.

XSD Generated Types: Element Type Instances Compared to Backing Type Instances

Suppose you have a XSD that defines one phone number simple type and multiple elements that use that simple type.

The XSD might look like the following:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="person">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="cell" type="phone"/>
                <xsd:element name="work" type="phone"/>
                <xsd:element name="home" type="phone"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:complexType name="phone">
      <xsd:sequence>
        <xsd:element name="areaCode" type="xsd:string"/>
        <xsd:element name="mainNumber" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Suppose you want to create and assign the phone numbers. There are multiple ways to do this.

If you want to create three different phone numbers, use code like this:

```
var e = new schema.Person()

e.Cell.AreaCode = "415"
e.Cell.MainNumber = "555-1213"

e.Work.AreaCode = "416"
e.Work.MainNumber = "555-1214"

e.Home.AreaCode = "417"
```

```
e.Home.MainNumber = "555-1215"
```

In contrast, you want to create one phone number to use in multiple elements, you might use code like this:

```
var e = new schema.Person()

var p = new schema.types.complex.Phone()
p.AreaCode = "415"
p.MainNumber = "555-1212"

e.Cell.$TypeInstance = p
e.Work.$TypeInstance = p
e.Home.$TypeInstance = p
```

An element's `$TypeInstance` property accesses the element's *backing type instance*.

It is important to note that it is necessary to use the `$TypeInstance` property syntax because the Gosu declared types of each phone number element are incompatible.

For example, you cannot create the complex type and directly assign it to the element type:

```
var e = new schema.Person()

var p = new schema.types.complex.Phone()
p.AreaCode = "415"
p.MainNumber = "555-1212"

e.Cell = p // SYNTAX ERROR: cannot assign complex type instance to element type instance
e.Work = p // SYNTAX ERROR: cannot assign complex type instance to element type instance
e.Home = p // SYNTAX ERROR: cannot assign complex type instance to element type instance
```

Additionally, different element-based types can be **mutually incompatible for assignment** even if they are associated with the XSD type definition. For example:

```
var e = new schema.Person()

e.Cell = e.Work // SYNTAX ERROR: cannot assign one element type to a different element type
```

Automatic Insertion into Lists

If you are using XSDs, for properties that represent child elements that can appear more than once, Gosu exposes that property as a list. For properties that Gosu exposes as list properties (see "XSD-based Properties and Types" on page 204), Gosu has a special shorthand syntax for **inserting** items into the list. If you assign to the list index equal to the size of the list, then the index assignment becomes an insertion.

This is also true if the size of the list is zero: use the `[0]` array/list index notation and set the property. This inserts the value into the list, which is equivalent to adding an element to the list. However, you do not have to worry about whether the list exists yet if you use this syntax. (If you are creating XML objects in Gosu, by default the lists do not yet exist. From Gosu they are `null`.)

In other words, you can add an element with the syntax:

```
element.PropertyName[0] = childElement
```

If the list does not exist yet for a list property at all, Gosu **creates** the list upon the first insertion.

In other words, suppose an element contains child elements that represent an address and the child element has the name `Address`. If the XSD declares the element could exist more than once, the `element.Address` property is a list of addresses. The following code creates a new `Address`:

```
element.Address[0] = new my.package.xsdname.elements.Address()
```

IMPORTANT If you use XSDs, Gosu automatically creates intermediate XML elements as needed. Use this feature to significantly improve the readability of your XML-related Gosu code.

For example, suppose you have 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" type="xsd:int" maxOccurs="unbounded"/>
```

```
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Run the following 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()
```

This outputs:

```
Before insertion: 0
After insertion: 3
<?xml version="1.0"?>
<Element1>
  <Child1>0</Child1>
  <Child1>1</Child1>
  <Child1>2</Child1>
</Element1>
```

This also works with simple types derived by list (`xsd:list`):

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Element1">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="Child1">
                    <xsd:simpleType>
                        <xsd:list itemType="xsd:int"/>
                    </xsd:simpleType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Output after running the exact same Gosu code:

```
Before insertion: 0
After insertion: 3
<?xml version="1.0"?>
<Element1>
  <Child1>0 1 2</Child1>
</Element1>
```

XSD List Property Example

As mentioned earlier in this topic, if the possibility exists for a child element name to appear multiple times, then the property becomes a list-based property.

For example, consider the following 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>
```

The following code uses this XSD:

```
var xml = new schema.Element1()
xml.Child1 = 1
xml.print()

print( "----------" )

xml.Child1 = null
```

```
xml.Child2 = { 1, 2, 3, 4 }
xml.print()
```

This prints the following:

```
<?xml version="1.0"?>
<Element1>
  <Child1>1</Child1>
</Element1>
----------
<?xml version="1.0"?>
<Element1>
  <Child2>1</Child2>
  <Child2>2</Child2>
  <Child2>3</Child2>
  <Child2>4</Child2>
</Element1>
```

Getting Data From an XML Element

The main work of an XML element happens in the type instance associated with each XML element. (For more information about this, see "Introduction to XmlElement" on page 196.) The type instance of an XML element is responsible for nearly all of the element behavior but does not contain the element's name. You can usually ignore the division of labor between an `XmlElement` and its backing type instance. If you are using an XSD, this distinction is useful.

If you instantiate a type instance, typically you use more specific subclass of `gw.xsd.w3c.xmlschema.types.complex.AnyType`.

Gosu exposes properties and methods on the XML type instances for you to get child elements or simple value.

It is important to note that XML elements contain two basic types of content:

- child elements
- simple value

An element can contain either child elements or a simple value, but not both at the same time.

IMPORTANT Elements contain child elements or text, but never both at the same time.

Manipulating Elements and Values (Works With or Without XSD)

To **get the child elements** of an element, get its `Children` property. The `Children` property contains a list (`java.util.List<XmlElement>`) of elements. If this XML element is an XSD-based type, you must add the property name prefix `$`, so instead get the property called `$Children`.

If the element has no child elements, there are two different cases:

- If an element has no child elements and no text content, the `Children` property contains an **empty list**.
- If an element has no child elements but has text content, the `Children` property contains `null`.

To **add a child element**, call the parent element's `addChild` method and pass the child element as a parameter.

For example, suppose you had the following XSD:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Element1">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="Child1"/> <!-- default type is xsd:anyType -->
                <xsd:element name="Child2" type="xsd:int"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Things to notice in this XSD:

- the element named `Child1` has no explicit type. This means the default type applies, which is `xsd:anyType`.
- the element named `Child2` has the type `xsd:int`. This means that by definition, this element must contain an integer value. Integer is a simple type. Without the integer value (if it were empty or `null`), any XML for this document would be invalid according to the XSD.

If you have a reference to an XML element of a simple type, you can set its value by setting its `SimpleValue` property. (If you are using an XSD, add the dollar sign prefix: `$SimpleValue`)

To **set a simple value** (like an integer value for an element), there are several approaches:

- Set the value in the `SimpleValue` property, to a subclass of `XmlSimpleValue`. This allows you to directly create the simple value that Gosu stores in the pre-serialized graph of XML elements. If it is on an XSD type, specify the property name with the dollar sign prefix: `$SimpleValue`. To create an instance of the `XmlSimpleValue` of the appropriate type, call static methods on the `XmlSimpleValue` type with method names that start with `make...`. For example, call the `makeIntInstance` method and pass it an `Integer`. It returns an `XmlSimpleValue` instance that represents an integer, and internally contains an integer. In memory, Gosu stores this information as a non-serialized value. Only during serialization of the XML, such as exporting into a byte array or using the debugging `print` method, does Gosu serialize the `XmlSimpleValue` into bytes or encoded text. For a full reference of all the simple value methods and all their variants, see “Simple Values” on page 214.
- To create simple text content (text simple value), set the element’s `Text` property to a `String` value. If it is on an XSD type, specify the property name with the dollar sign prefix: `$Text`.
- If you are using an XSD, you can set the natural value in the `Value` property. If it is on an XSD type, specify the property name with the dollar sign prefix: `$Value`. For example, use natural-looking code like `e.$Value = 5`. If you are using an XSD and have non-text content, this approach tends to result in more natural-looking Gosu code than creating instances of `XmlSimpleValue`.
- If you are using an XSD, Gosu provides a simple syntax to get and set child values with simple types. For example, set numbers and dates from an element’s parent element using natural syntax using the **child element name** as a property accessor. This lets you easily access the child element’s simple value with very readable code. For example, `e.AutoCost = 5`. See “XSD-based Properties and Types” on page 204.

The following Gosu code adds two child elements, sets the value of an element using the `Value` property and the `SimpleValue` property, and then prints the results. In this example, we use XSD types, so we must specify the special property names with the dollar sign prefix: `$Value` and `$SimpleValue`.

```
uses gw.xml.XmlSimpleValue

// create a new element, whose type is in the namespace of the XSD
var e =  new com.guidewire.pl.docexamples.gosu.xml.demochildprops.Element1()
var c = e.$Children   // returns an empty list of type List<XmlElement>
print("Children " + c.Count + c)
print("")

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

// create a new CHILD element that is legal in the XSD, and add it as child
var c2 = new com.guidewire.pl.docexamples.gosu.xml.demochildprops.anonymous.elements.Element1_Child2()
print("before set: " + c2.$Value)  // prints "null" -- it is uninitialized

c2.$SimpleValue = XmlSimpleValue.makeIntInstance(5)
print("after set with $SimpleValue: " + c2.$Value)

c2.$Value = 7
print("after set with $Value: " + c2.$Value)
print("")

// add the child element
e.addChild(c2)

c = e.$Children   // returns a list of two child elements
print("Children " + c.Count + c)

print("")
e.print()
```

This code prints the following:

```
Children 0[]

before set: null
after set with $SimpleValue: 5
after set with $Value: 7

Children 2[com.guidewire.pl.docexamples.gosu.xml.demochildprops.anonymous.elements.Element1_Child1
  instance, com.guidewire.pl.docexamples.gosu.xml.demochildprops.anonymous.elements.Element1_Child2
  instance]

<?xml version="1.0"?>
<Element1>
  <Child1/>
  <Child2>7</Child2>
</Element1>
```

Note that the `Child2` element contains the integer as text data in the serialized XML export. Gosu does not serialize the simple types to bytes (or a `String`) until serialization. In this example, the final `print` statement is what serializes the element and all its subelements.

Getting Child Elements By Name

If you want to iterate across the `List` of child elements to find your desired data, you can do so using the `Children` property mentioned earlier in this topic. Depending on what you are doing, you might want to use the Gosu enhancements on lists to find the items you want. See “Collections” on page 183 for more details.

However, it is common to want to get a child element **by its name**. To support this common case, Gosu provides methods on the XML element object. There are two main variants of this method. Use `getChild` if you expect only one match. Use `getChildren` if you expect multiple matches. Each one of these has an alternate signature that takes a `String`.

- `getChild( QName )` - searches the content list for a single child with the specified `QName` name. There is an alternate method signature that takes a `String` value for the local part name. For that method signature, Gosu 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. There is an alternate method signature that takes a `String` value for the local part name. For that method signature, Gosu internally creates a `QName` with an empty namespace and the specified local part name.

Reusing the code from the previous example, you could add the following lines to get the second child element by its name:

```
// Get a child using the empty namespce by passing a String
var getChild1 = e.getChild("Child1")

// Get a child using a QName, and "reuse" the namespace of a previous node
var getChild2FromQName = e.getChild(getChild1.Namespace.qualify("Child2"))

print(getChild2FromQName.asUTFString())
```

This 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. There is an alternate method signature that takes a `String` value for the local part name. For that method, Gosu internally creates a `QName` with an empty namespace and the specified local part name.

- `removeChildren( QName ) : List<XmlElement>` - Removes the child with the specified `QName` name. There is an alternate method signature that takes a `String` value for the local part name. For that method, Gosu internally creates a `QName` with an empty namespace and the specified local part name.

Attributes

Attributes are additional metadata on an element. For example, in the following example an element has the `color` and `size` attributes:

```
<myelement color="blue" size="huge">
```

Every type instance contains its attributes, which are `XmlSimpleValue` instances specified by a name (a QName). For more information about simple values, see “Simple Values” on page 214.

Each `XmlElement` object contains the following methods and properties

- `AttributeNames` property - Gets a set of QName objects. The property type is `java.util.Set<QName>`.
- `getAttributeSimpleValue( QName )`- Get attribute simple value by its name, specified as a QName. Returns a `XmlSimpleValue` object. There is an alternate method signature that takes a `String` instead of a QName, and it assumes an empty namespace.
- `getAttributeValue( QName ) : String` - Get attribute value by its name, specified as a QName. Returns a `String` object. There is an alternate method signature that takes a `String` instead of a QName, and it assumes an empty namespace.
- `setAttributeSimpleValue( QName , XmlSimpleValue )` - Set attribute simple value by its name (as a QName) and its value (as a `XmlSimpleValue` object). There is an alternate method signature that takes a `String` instead of a QName, and it assumes an empty namespace.
- `setAttributeValue( QName , String )` - Set attribute value by its name (as a QName) and its value (as a `XmlSimpleValue` object). There is an alternate method signature that takes a `String` instead of a QName, and it assumes an empty namespace.

Using the previous example, the following code gets and sets the attributes:

```
myelement.setAttributeValue("color", XmlSimpleValue.makeStringInstance("blue"))
var s = myelement.getAttributeValue("size")
```

Generally speaking, if you use XSDs for your elements, for typical use do not use these APIs. Instead, use the shortcuts that Gosu adds. They provide a natural and concise syntax for getting and setting attributes. For details, see “XSD-based Properties and Types” on page 204.

Simple Values

Gosu represents the XML format simple values with the `gw.xml.XmlSimpleValue` type. An `XmlSimpleValue` is a Gosu object that encapsulates a value and the logic to serialize that value to XML. However, until serialization occurs, Gosu may internally store it in a format other than `java.lang.String`.

IMPORTANT For more information about the role of simple values in Gosu XML APIs, see “Getting Data From an XML Element” on page 211.

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 when 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 GosuValue
- `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. This may not be the string that is actually serialized (such as in the case of a QName)

The following table lists static methods on the `XmlSimpleValue` type that create `XmlSimpleValue` instances of various types.

Method signature	Description
`makeStringInstance( java.lang.String )`	Make String instance
`makeAnyURIInstance( java.net.URI )`	Make URI instance
`makeBooleanInstance( java.lang.Boolean )`	Make boolean instance
`makeByteInstance( java.lang.Byte )`	Make byte instance
`makeUnsignedByteInstance( java.lang.Short )`	Make unsigned byte instance
`makeDateInstance( gw.xml.date.XmlDate )`	Make date-time instance from an `XmlDate`
`makeDateTimeInstance( gw.xml.date.XmlDateTime )`	Make date instance from an `XmlDateTime`
`makeDecimalInstance( java.math.BigDecimal )`	make decimal instance from a `BigDecimal`
`makeDoubleInstance( java.lang.Double )`	make decimal instance from a Double
`makeDurationInstance( gw.xml.date.XmlDuration )`	Make duration instance
`makeFloatInstance( java.lang.Float )`	Make float instance
`makeGDayInstance( gw.xml.date.XmlDay )`	Make `GDay` instance
`makeGMonthDayInstance( gw.xml.date.XmlMonthDay )`	Make `GMonthDay` duration instance
`makeGMonthInstance( gw.xml.date.XmlMonth )`	Make `GMonth` instance
`makeGYearInstance( gw.xml.date.XmlYear )`	Make `GYear` instance
`makeGYearMonthInstance( gw.xml.date.XmlYearMonth )`	Make `GYearMonth` instance
`makeHexBinaryInstance( byte[] )`	Make hex binary instance from byte array
`makeIDInstance( java.lang.String )`	Make `IDInstance` instance from a String
`makeIDREFInstance( gw.xml.XmlElement )`	Make `IDREF` instance
`makeIntegerInstance( java.math.BigInteger )`	Make big integer instance
`makeIntInstance( java.lang.Integer )`	Make integer instance
`makeLongInstance( java.lang.Long )`	Make long integer instance
`makeUnsignedIntInstance( java.lang.Long )`	Make unsigned integer instance
`makeUnsignedLongInstance( java.math.BigInteger )`	Make unsigned long integer instance
`makeQNameInstance( javax.xml.namespace.QName )`	Make `QName` instance
`makeQNameInstance(java.lang.String, javax.xml.namespace.NamespaceContext )`	Make `QName` instance from a standard Java namespace context. A namespace context object encapsulates a mapping of XML namespace prefixes and their definitions (namespace URIs). You can get an instance of `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 )`	Make duration instance
`makeUnsignedShortInstance( java.lang.Integer )`	Make unsigned short integer instance
`makeTimeInstance( gw.xml.date.XmlTime )`	Make duration instance
`makeBase64BinaryInstance( byte[] )`	Make base 64 binary instance from byte array
`makeBase64BinaryInstance( gw.xml.BinaryDataProvider )`	Make base 64 binary instance from binary data provider

XSD to Gosu Simple Type Mappings

For all elements with simple types and all attributes in an XSD, Gosu creates properties based on which simple schema type it is. The following table describes how Gosu maps XSD schema types to Gosu types. For schema types that are not listed in the table, Gosu uses the schema type's supertype. For example, the schema type `String` is not listed, so Gosu uses its supertype `anySimpleType`.

Schema Type	Gosu Type
`boolean`	`java.lang.Boolean`
`byte`	`java.lang.Byte`
`decimal`	`java.math.BigDecimal`
`double`	`java.lang.Double`
`float`	`java.lang.Float`
`int`	`java.lang.Integer`
`integer`	`java.math.BigInteger`
`long`	`java.lang.Long`
`short`	`java.lang.Short`
`unsignedLong`	`java.math.BigInteger`
`unsignedInt`	`java.lang.Long`
`unsignedShort`	`java.lang.Integer`
`unsignedByte`	`java.lang.Short`
`date`	`gw.xml.date.XmlDate`
`dateTime`	`gw.xml.date.XmlDateTime`
`time`	`gw.xml.date.XmlTime`
`gYearMonth`	`gw.xml.date.XmlYearMonth`
`gYear`	`gw.xml.date.XmlYear`
`gMonthDay`	`gw.xml.date.XmlMonthDay`
`gDay`	`gw.xml.date.XmlDay`
`gMonth`	`gw.xml.date.XmlMonth`
`duration`	`gw.xml.date.XmlDuration`
`base64Binary`	`gw.xml.BinaryDataProvider`
`hexBinary`	`byte[]`
`anyURI`	`java.net.URI`
`QName`	`javax.xml.namespace.QName`
`IDREF`	`gw.xml.XmlElement`
`anySimpleType`	`java.lang.String`
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 runtime. This mostly affects facets relating to lengths of lists, and those that validate QNames. 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 QNames at all.

Example:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Element1">
        <xsd:complexType>
            <xsd:attribute name="Attr1" type="AttrType"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:simpleType name="AttrType">
        <xsd:restriction base="xsd:int">
            <xsd:minInclusive value="0"/>
            <xsd:maxInclusive value="5"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

Code:

```
var xml = new schema.Element1()
xml.Attr1 = 3 // works
xml.Attr1 = 6 // fails with exception
```

Output:

```
gw.xml.XmlSimpleValueException: Value '6' violated one or more facet constraints
of simple type definition: value must be no greater than 5
```

Access the Nillness of an Element

XML has a concept of whether an element is `nil`. This is not exactly same as being `null`. An element can be `nil` (and must have no child elements) but *still have attributes*. Additionally, an XSD can define whether an element is *nillable*, which means that element is allowed to be `nil`.

If an XSD-based element is nillable, the `XmlElement` object exposes a property with the name `$Nil`. All non-XSD elements also have this property, but it is called `Nil` (with no dollar sign prefix). Note that nillability is an XSD concept, so for non-XSD elements the element can always *potentially* be `nil`.

WARNING 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`).

Note that nillability is an aspect of elements (and only XSD-based elements), rather than an aspect of the XSD type itself. For more on the distinction between `XmlElement` and its backing type, see “Introduction to XmlElement” on page 196 in the *Gosu Reference Guide*

For example, consider the following XSD:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Element1" type="xsd:int" nillable="true"/>
</xsd:schema>
```

Consider the following code:

```
var xml = new schema.Element1()
xml.$Nil = true
xml.print()
```

This prints:

```
<?xml version="1.0"?>
<Element1 xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
```

Automatic Creation of Intermediary Elements

When using XSDs, when a property path takes part in the left hand side of an assignment statement, Gosu creates any intermediary elements to ensure the assignment from Gosu works. This is a very useful shortcut. Use this feature to make your Gosu code significantly more understandable.

> **IMPORTANT** If you use XSDs, Gosu automatically creates intermediate XML elements as needed. Use this feature to significantly improve the readability of your XML-related Gosu code.

For example, consider 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:complexType>
                        <xsd:sequence>
                            <xsd:element name="Child2" type="xsd:int"/>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Run the following code:

```
var xml = new schema.Element1()
print( "Before assignment: ${xml.Child1}" )
xml.Child1.Child2 = 5
print( "After assignment: ${xml.Child1}" )
```

This prints:

```
Before assignment: null
After assignment: schema.anonymous.elements.Element1_Child1 instance
```

Default/Fixed Attribute Values

The defaulting of `default` and `fixed` attribute/element values is solely an aspect of the statically typed property getter for that element or attribute. These values are not actually stored in the type instance's attribute map or content list, nor are they serialized, unless explicit. Keep in mind that elements pick up their default/fixed values if and only if they exist (per the XML Schema specification).

Example:

```
<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>
```

Code:

```
var xml = new schema.Root()
xml.Person[0].Name = "jsmith"
xml.Person[0].Os = Linux
xml.Person[1].Name = "aanderson"
for ( person in xml.Person ) {
  print( "${person.Name} (${person.Location}) -> ${person.Os}" )
}
xml.print()
```

Output:

```
jsmith (San Mateo) -> Linux
aanderson (San Mateo) -> Windows
<?xml version="1.0"?>
<root>
  <person os="Linux">
    <name>jsmith</name>
  </person>
  <person>
    <name>aanderson</name>
  </person>
</root>
```

Substitution Group Hierarchies

Just as Gosu reproduces XSD-defined type hierarchies in the Gosu type system, Gosu also exposes XSD-defined substitution group hierarchies.

The name *substitution group* is the standard name for this XSD feature, although the name can be somewhat confusing. An XSD `substitutionGroup` attribute can be defined on any top-level element to indicate the QName of another top-level element that it can be substituted for. The name *substitution group* comes from its normal use, which is to create a *substitution group head* (the group's main element) with some abstract name, such as `"Address"`.

Next, create *substitution group members*. To create a substitution group member, set the XML attribute `substitutionGroup` on an element to the element name (QName) of the *substitution group head*.

There is no need to indicate at runtime that the substitution happened place. This is in contrast to subtypes, in which `xsi:type` must be present. If an XML element uses a substitution group member QName in place of the head's QName, the Gosu XML processor knows that the substitution happened.

Example:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xsd:element name="Customer">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="Address"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="Address"/>
    <xsd:element name="USAddress" substitutionGroup="Address"/>
    <xsd:element name="UKAddress" substitutionGroup="Address"/>
</xsd:schema>
```

Code:

```
var xml = new schema.Customer()
xml.Address = new schema.UKAddress()
xml.print()
```

Output:

```
<?xml version="1.0"?>
<Customer>
  <UKAddress/>
</Customer>
```

The XML Schema specification requires that the XSD type of a substitution group member must be a subtype of the XSD type of its substitution group head. The reason the example above works is because `UKAddress`, `USAddress` and `Address` are all of the type `xsd:anyType` (the default when there is no explicit type).

Element Sorting for XSD-based Elements

An XSD can define the strict order of children of an element. For non-XSD elements, element order is undefined.

Each `XmlElement` exposes a `Children` property. (For XSD-based elements the property name is `$Children`.)

If the list of child elements is out of order according to the XSD, Gosu sorts the element list during serialization to match the schema. This sorting does not affect the original order of the elements in the content list.

If you use APIs to directly add child elements, such as adding to the child element list or using an `addChild` method, you can add child elements out of order. Similarly, some APIs indirectly add child elements, such as such as autocreation of intermediary elements (see “Automatic Creation of Intermediary Elements” on page 218.). In all of these cases, Gosu permits the children to be out of order in the `XmlElement` object graph.

During serialization (and **only** during serialization), Gosu sorts the elements to ensure that the elements conform to the XSD.

Note that if you parse XML into an `XmlElement` using an XSD, the elements must be in the correct order according to the XSD. If the child order violates the XSD, Gosu throws an exception during parsing.

Example XSD:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Element1">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="Child1" type="xsd:int"/>
                <xsd:element name="Child2" type="xsd:int"/>
                <xsd:element name="Child3" type="xsd:int"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Code:

```
var xml = new schema.Element1()
xml.Child2 = 2
xml.Child1 = 1
xml.Child3 = 3
xml.print()
```

Output:

```
<?xml version="1.0"?>
<Element1>
  <Child1>1</Child1>
  <Child2>2</Child2>
  <Child3>3</Child3>
</Element1>
```

Another example XSD:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Element1">
        <xsd:complexType>
            <xsd:choice>
                <xsd:sequence>
                    <xsd:element name="A" type="xsd:int"/>
                    <xsd:element name="B" type="xsd:int"/>
                    <xsd:element name="C" type="xsd:int"/>
                </xsd:sequence>
                <xsd:sequence>
                    <xsd:element name="B" type="xsd:int"/>
                    <xsd:element name="C" type="xsd:int"/>
                    <xsd:element name="A" type="xsd:int"/>
                    <xsd:element name="Q" type="xsd:int"/>
                </xsd:sequence>
            </xsd:choice>
```

```
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Code:

```
var xml = new schema.Element1()
xml.A = 5
xml.B = 5
xml.C = 5
xml.print()
print( "----------" )
xml.Q = 5
xml.print()
```

Output:

```
<?xml version="1.0"?>
<Element1>
  <A>5</A>
  <B>5</B>
  <C>5</C>
</Element1>
----------
<?xml version="1.0"?>
<Element1>
  <B>5</B>
  <C>5</C>
  <A>5</A>
  <Q>5</Q>
</Element1>
```

If Element Order Is Already Correct

If the children of an element are in an order that matches the XSD, Gosu does not sort the element list. This is important if there is more than sorted order that conforms to the XSD and you desire a particular order.

For 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>
```

Code:

```
var xml = new schema.Element1()
xml.A = 5
xml.B = 5
xml.C = 5
xml.print()

print( "----------" )

xml = new schema.Element1()
xml.C = 5
xml.B = 5
xml.A = 5
xml.print()
```

Output:

```
<?xml version="1.0"?>
<Element1>
  <A>5</A>
  <B>5</B>
```

```
  <C>5</C>
</Element1>
----------
<?xml version="1.0"?>
<Element1>
  <C>5</C>
  <B>5</B>
  <A>5</A>
</Element1>
```

Multiple Correct Sort Order Matches

If the children of an element are out of order, but multiple correct orderings exist, the first correct ordering defined in the schema will be used.

Example XSD:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Element1">
        <xsd:complexType>
            <xsd:choice>
                <xsd:sequence>
                    <xsd:element name="A" type="xsd:int"/>
                    <xsd:element name="B" type="xsd:int"/>
                    <xsd:element name="C" type="xsd:int"/>
                </xsd:sequence>
                <xsd:sequence>
                    <xsd:element name="C" type="xsd:int"/>
                    <xsd:element name="B" type="xsd:int"/>
                    <xsd:element name="A" type="xsd:int"/>
                </xsd:sequence>
            </xsd:choice>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Code:

```
var xml = new schema.Element1()
xml.C = 5
xml.A = 5
xml.B = 5
xml.print()
```

Output:

```
<?xml version="1.0"?>
<Element1>
  <A>5</A>
  <B>5</B>
  <C>5</C>
</Element1>
```

Built-in Schemas

Gosu includes several XSDs in the `gw.xsd.*` package. The following table lists the built-in XSDs.

Description of the XSD	Fully-qualified package name for the XSD
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. See "The XSD that Defines an XSD (The Metaschema)" on page 223.	`gw.xsd.w3c.xmlschema.Schema`

The XSD that Defines an XSD (The Metaschema)

The definition of an XSD is itself an XML file. The *XML Schema XSD* is the XSD that defines the XSD format. It is also known as the *metaschema*. It is in the Gosu package `gw.xsd.w3c.xmlschema`. This schema is sometimes useful for building or parsing schemas.

Example:

```
var schema = new gw.xsd.w3c.xmlschema.Schema()
schema.Element[0].Name = "Element1"
schema.Element[0].ComplexType.Sequence.Element[0].Name = "Child"
schema.Element[0].ComplexType.Sequence.Element[0].Type = new javax.xml.namespace.QName( "Type1" )
schema.ComplexType[0].Name = "Type1"
schema.print()
```

Output:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Element1">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Child" type="Type1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="Type1"/>
</xsd:schema>
```

There is no way to inject a schema into the type system at run time.

Schema Access Type

For each XSD that Gosu loads, it creates a `SchemaAccess` object that represents the loaded XSD.

The most important reason to use this is to includes additional schemas during XML parsing. For more information, see “Parsing XML Data into an XML Element” on page 201.

Additionally, you can get this object’s `Schema` property, which is the Gosu XML representation of the XSD as an XML file. In other words, this contains the `gw.xsd.w3c.xmlschema.Schema` object that represents this XSD.

For 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>
```

Run this code:

```
var schema = schema.util.SchemaAccess.Schema
schema.Element.each( \ el ->print( el.Name ) )
```

This prints:

```
Element1
Element2
Element3
```

The following example uses the XSD of XSDs (see “The XSD that Defines an XSD (The Metaschema)” on page 223) to print a list of primitive schema types:

```
var schema = gw.xsd.w3c.xmlschema.util.SchemaAccess.Schema
print( schema.SimpleType.where( \ s ->s.Restriction.Base.LocalPart == "anySimpleType" ).map(
   \ s ->s.Name ) )
```

Output:

```
[string, boolean, float, double, decimal, duration, dateTime, time, date, gYearMonth,
gYear, gMonthDay, gDay, gMonth, hexBinary, base64Binary, anyURI, QName, NOTATION]
```

The APIs described in this topic generate the entire XML graph.

chapter 19

Calling WS-I Web Services from Gosu

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

This topic includes:

Consuming WS-I Web Service Overview

Gosu supports calling WS-I compliant web services. WS-I is an open industry organization that promotes industry-wide best practices for web services interoperability among diverse systems. The organizations provides several different profiles and standards. The WS-I Basic Profile is the baseline for interoperable web services and more consistent, reliable, and testable APIs.

Gosu offers native WS-I web service client code with the following features:

- Call web service methods with natural Gosu syntax for method calls
- Call web services optionally asynchronously. See "Asynchronous Methods" on page 233.
- Support one-way web service methods. See "One-Way Methods" on page 233.
- Separately encrypt requests and responses. See "Adding WS-I Configuration Options" on page 230.
- Sign incoming responses with digital signatures. See "Implementing Advanced Web Service Security with WSS4J" on page 231.

One of the big differences between WS-I and older styles of web services is how the client and server encodes API parameters and return results.

An older style of web services is called *Remote Procedure Call encoded* (RPCE) web services. The bulk of the incoming and outgoing data are encoded in a special way that does not conform to XSD files. Many older systems use RPCE web services, but there are major downsides with this approach. Most notably, the encoding is specific to remote procedure calls, so it is difficult to validate XML data in RPC encoded responses. It would be more convenient to use standard XML validators which rely on XSDs to define the structure of the main content.

When you use the WS-I standards, you can use the alternative encoding called Document Literal encoding (`document/literal`). The document-literal-encoded SOAP message contains a complete XML document for each method argument and return value. The schema for each of these documents is an industry-standard XSD file. The WSDL that describes how to talk to the published WS-I service includes a complete XSD describing the format of the embedded XML document. The outer message is very simple, and the inner XML document contains all of the complexity. Anything that an XSD can define becomes a valid payload or return value.

The WS-I standard supports a mode called RPC Literal (`RPC/literal`) instead of Document Literal. Despite the similarity in name, WS-I RPC Literal mode is not closely related to RPC encoding. Gosu supports this WS-I RPC Literal mode for Gosu web service client code. However, it does so by automatically and transparently converting any WSDL for RPC Literal mode into WSDL for Document Literal mode. The focus of the Gosu documentation for WS-I web services is the support for Document Literal encoding.

Loading WS-I WSDL Directly into the File System

To consume an external web service, you must load the WSDL and XML schema files (XSDs) for the web service. You must fetch copies of WSDL files, as well as related WSDL and XSD files, from the web services server. Fetch the copies into an appropriate place in the Gosu class hierarchy on your local system. Place them in the directory that corresponds to the package in which you want new types to appear. For example, place the WSDL and XSD files next to the Gosu class files that call the web service, organized in package hierarchies just like class files.

not from the WSDLThe following sample Gosu code shows how to manually fetch web service WSDLs for test purposes or for command-line use from a web service server.

```
uses gw.xml.ws.*
uses java.net.URL
uses java.io.File

// -- set the web service endpoint URL for the web service WSDL --
var urlStr = "http://www.aGreatWebService.com/GreatWebService?wsdl"

// -- set the location in your file system for the web service WSDL --
var loc = "/wsi/remote/GreatWebService"

// -- load the web service WSDL into Gosu --
Wsdl2Gosu.fetch(new URL(urlStr), new File(loc))
```

The first long string (`urlStr`) is the URL to the web service WSDL. The second long string (`loc`) is the path on your file system where the fetched WSDL is stored. You can run your version of the preceding sample Gosu code in the Gosu Tester.

Security and Authentication

The WS-I basic profile requires support for some types of security standards for web services, such as encryption and digital signatures (cryptographically signed messages). See “Adding WS-I Configuration Options” on page 230.

Types of WS-I Client Connections

From Gosu, there are three types of WS-I web service client connections:

- Standard round trip methods (synchronous request and response)

- Asynchronous round trip methods (send the request and immediately return to the caller, and check later to see if the request finished). See “Asynchronous Methods” on page 233.
- One-way methods, which indicate a method defined to have no SOAP response at all. See “One-Way Methods” on page 233.

How Does Gosu Process WSDL?

Suppose you add a WSDL file directly to your class hierarchy called `MyService.wsdl` in the package `example.pl.gs.wsic`. Gosu creates all the types for your web service in the namespace:

```
example.pl.gs.wsic.myservice.*
```

The name of `MyService` becomes lowercase `myservice` in the package hierarchy for the XML objects because the Gosu convention for package names is lowercase. There are other name transformations as Gosu imports types from XSDs. For details, see “Normalization of Gosu Generated XSD-based Names” on page 207.

The structure of a WSDL comprises the following:

- One or more **services**
- For each **service**, one or more **ports**

  A port represents a protocol or other context that might change how the WSDL defines that service. For example, methods might be defined differently for different versions of SOAP, or an additional method might be added for some ports. WSDL might define one port for SOAP 1.1 clients, one port for SOAP 1.2 clients, one port for HTTP non-SOAP access, and so on. See discussion later in this topic for what happens if multiple ports exist in the WSDL.
- For each **port**, one or more **methods**

  A method, also called an operation or action, performs a task and optionally returns a result. The WSDL includes XML schemas (XSDs), or it imports other WSDL or XSD files. Their purposes are to describe the data for each *method argument type* and each *method return type*.

Suppose the WSDL looks like the following:

```
<wsdl>
  <types>
    <schema>
      <import schemaLocation="yourschema.xsd"/>
      <!-- now define various operations (API methods) in the WSDL ... -->
```

The details of the web service APIs are omitted in this example WSDL. Assume the web service contains exactly one service called `SayHello`, and that service contains one method called `helloWorld`. Let us assume for this first example that the method takes no arguments, returns no value, and is published with no authentication or security requirements.

In Gosu, you can call the remote service represented by the WSDL using code such as:

```
// get a reference to the service in the namespace of the
var service = new example.pl.gs.wsic.myservice.SayHello()

// call a method on the service
service.helloWorld()
```

Of course, real APIs need to transfer data also. In our example WSDL, notice that the WSDL refers a secondary schema called `yourschema.xsd`.

Add any additional XSD into the `*web_service_name*.wsdl.resources` subdirectory.

Let us suppose the contents of your `yourschema.xsd` file looks like the following:

```
<schema>
  <element name="Root" type="xsd:int"/>
</schema>
```

Note that the element name is `"root"` and it contains a simple type (`int`). This XSD represents the format of an element for this web service. The web service could declare a `<root>` element as a method argument or return type.

Now let us suppose there is another method in the SayHello service called doAction and this method takes one argument that is a <root> element.

In Gosu, you can call the remote service represented by the WSDL using code similar to the following:

```
// get a reference to the service
var service = new ws.myservice.SayHello()

// create an XML document from the WSDL using the Gosu XML API
var x = new ws.myservice.Root()

// call a method that the web service defines
var ret = service.doAction( x )
```

The package names are different if you place your WSDL file in a different part of the package hierarchy.

For each web service API call, Gosu first evaluates the method parameters. Internally, Gosu serializes the root XmlElement instance and its child elements into XML raw data using the associated XSD data from the WSDL. Next, Gosu sends the resulting XML document to the web service. In the SOAP response, the main data is an XML document, whose schema is contained in (or referenced by) the WSDL.

Learning Gosu XML APIs

All WS-I method argument types and return types are defined from schemas (the XSD embedded in the WSDL). From Gosu, all these objects are instances of subclasses of XmlElement, with the specific subclass defined by the schemas in the WSDL. From Gosu, working with WS-I web service data requires that you understand Gosu XML APIs.

In many cases, Gosu hides much of the complexity of XML so you do not need to worry about it. For example, for XSD-types, in Gosu you do not have to directly manipulate XML as bytes or text. Gosu handles common types like number, date, or Base64 binary data. You can directly get or set values (such as a Date object rather than a serialized xsd:date object). The XmlElement class, which represents an XML element hide much of the complexity.

Other notable tips to working with XML in Gosu:

- When using a schema (an XSD, or a WSDL that references an XSD), Gosu exposes shortcuts for referring to child elements using the name of the element. For more information, see “XSD-based Properties and Types” on page 204.
- When setting properties in an XML document, Gosu creates intermediate XML element nodes in the graph automatically. Use this feature to significantly improve the readability of your XML-related Gosu code. For details, see “Automatic Creation of Intermediary Elements” on page 218.
- For properties that represent child elements that can appear more than once, Gosu exposes that property as a list. For list-based types like this, there is a special shortcut to be aware of. If you assign to the list index **equal to the size of the list**, then Gosu treats the index assignment as an insertion. This is also true if the size of the list is zero: use the [0] array/list index notation and set the property. This inserts the value into the list, which is equivalent to adding an element to the list. However, you do not have to worry about whether the list exists yet. If you are creating XML objects in Gosu, by default the lists do not yet exist.

  In other words, use the syntax:

  ```
  element.PropertyName[0] = childElement
  ```

  If the list does not exist yet for a list property at all, Gosu **creates** the list upon the first insertion. In other words, suppose an element contains child elements that represent an address and the child element has the name Address. If the XSD declares the element could exist more than once, the element.Address property is a list of addresses. The following code creates a new Address:

  ```
  element.Address[0] = new my.package.xsdname.elements.Address()
  ```

For more information, see the related topics:

- “Automatic Insertion into Lists” on page 209
- “XSD-based Properties and Types” on page 204

- "Gosu and XML" on page 195

What Gosu Creates from Your WSDL

Within the namespace of the package of the WSDL, Gosu creates some new types.

For each service in the web service, Gosu creates a service by name. For example, if the external service has a service called `GeocodeService` and the WSDL is in the package `examples.gosu.wsi`, then the service has the fully-qualified type `examples.gosu.wsi.GeocodeService`. Create a new instance of this type, and you then you can call methods on it for each method.

For each operation in the web service, generally speaking Gosu creates two local methods:

- one method with the method name in its natural form, for example suppose a method is called `doAction`
- one method with the method name with the `async_` prefix, for example `async_doAction`. This version of the method handles asynchronous API calls. For details, see "Asynchronous Methods" on page 233.

Special Behavior For Multiple Ports

Gosu automatically processes ports from the WSDL identified as either SOAP 1.1 or SOAP 1.2. If both are available for any service, Gosu ignores the SOAP 1.1 ports. In some cases, the WSDL might define more than one available port (such as two SOAP 1.2 ports with different names).

For example, suppose you add a WSDL file to your class hierarchy called `MyService.wsdl` in the package `example.pl.gs.wsic`. Gosu chooses a default port to use and creates types for the web service at the following path:

```
ROOT_PACKAGE.WSDL_NAME_NORMALIZED.NORMALIZED_SERVICE_NAME
```

The *`NORMALIZED_SERVICE_NAME`* name of the package is the name of the service as defined by the WSDL, with capitalization and conflict resolution as necessary. For example, if there are two services in the WSDL named `Report` and `Echo`, then the API types are in the location

```
example.pl.gs.wsic.myservice.Report
example.pl.gs.wsic.myservice.Echo
```

Gosu chooses a default port for each service. If there is a SOAP 1.2 version, Gosu prefers that version.

Additionally, Gosu provides the ability to explicitly choose a port. For example, if there is a SOAP 1.1 port and a SOAP 1.2 port, you could explicitly reference one of those choices. Gosu creates all the types for your web service ports within the `ports` subpackage, with types based on the name of each port in the WSDL:

```
ROOT_PACKAGE.WSDL_NAME_NORMALIZED.ports.SERVICE_AND_PORT_NAME
```

The *`SERVICE_AND_PORT_NAME`* is the service name, followed by an underscore, followed by the port name.

For example, suppose the ports are called `p1` and `p2` and the service is called `Report`. Gosu creates types within the following packages:

```
example.pl.gs.wsic.myservice.ports.Report_p1
example.pl.gs.wsic.myservice.ports.Report_p2
```

Additionally, if the port name happens to begin with the service name, Gosu removes the duplicate service name before constructing the Gosu type. For example, if the ports are called `ReportP1`, and `helloP2`, Gosu creates types within the following packages:

```
example.pl.gs.wsic.myservice.ports.Report_P1       // NOTE: it is not Report_ReportP1
example.pl.gs.wsic.myservice.ports.Report_helloP2  // not a duplicate, so Gosu does not remove "Hello"
```

Each one of those hierarchies would include method names for that port for that service.

A Real Example: Weather

There is a public free web service that provides the weather. You can get the WSDL for this web service at the URL `http://ws.cdyne.com/WeatherWS/Weather.asmx?wsdl`. This web service does not require authentication or encryption.

The following Gosu code gets the weather in San Francisco:

```
var ws = new ws.weather.Weather()
var r = ws.GetCityWeatherByZIP(94114)
print( r.Description )
```

Depending on the weather, your result might be something like:

```
Mostly Sunny
```

Adding WS-I Configuration Options

If a web service does not need encryption, authentication, or digital signatures, you can just instantiate the service object and call methods on it:

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

// call a method on the service
api.helloWorld()
```

Directly Modifying the WSDL Configuration Object for a Service

To add authentication or security settings to a web service you can do so by modifying the options on the service object. To access the options from the API object (in the previous example, the object in the variable called `api`), use the syntax `api.Config`. That property contains the API options object, which has the type `gw.xml.ws.WsdlConfig`.

The WSDL configuration object has properties that add or change authentication and security settings. The `WsdlConfig` object itself is not an XML object (it is not based on `XmlElement`), but some of its subobjects are defined as XML objects. Fortunately, in typical code you do not need to really think about that difference. Instead, simply use a straightforward syntax to set authentication and security parameters. The following subtopics describe `WsdlConfig` object properties that you can set on the WSDL configuration object.

> **Note:** For XSD-generated types, if you set a property several levels down in the hierarchy, Gosu adds any intermediate XML elements if they did not already exist. This makes your XML-related code look concise. See also “Automatic Creation of Intermediary Elements” on page 218 in the *Gosu Reference Guide*.

HTTP Authentication

To add simple HTTP authentication to API request, use the basic HTTP authentication object at the path as follows. Suppose *`api`* is a reference to a SOAP API that you have already instantiated with the `new` operator. The properties on which to set the name and password are on the object:

```
api.Config.Http.Authentication.Basic
```

That object has a `Username` property for the user name and a `Password` property for the password. Set those two values with the desired user name and password.

For example:

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

service.Config.Http.Authentication.Basic.Username = "jms"
service.Config.Http.Authentication.Basic.Password = "b5"

// call a method on the service
service.helloWorld()
```

Setting a Timeout

To set the timeout value (in milliseconds), set the `CallTimeout` property on the `WsdlConfig` object for that API reference.

For example:

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

service.Config.CallTimeout = 30000 // 30 seconds

// call a method on the service
service.helloWorld()
```

Custom SOAP Headers

SOAP HTTP headers are essentially XML elements attached to the SOAP envelope for the web service request or its response. Your code might need to send additional SOAP headers to the external system, such as custom headers for authentication or digital signatures. You also might want to read additional SOAP headers on the response from the external system.

To add SOAP HTTP headers to a request that you initiate, first construct an XML element using the Gosu XML APIs (`XmlElement`). Next, add that `XmlElement` object to the list in the location *`api`*`.Config.RequestSoapHeaders`. That property contains a list of `XmlElement` objects, which in generics notation is the type `java.util.ArrayList<XmlElement>`.

To read (get) SOAP HTTP headers from a response, it only works if you use the asynchronous SOAP request APIs described in "Asynchronous Methods" on page 233. There is no equivalent API to get just the SOAP headers on the response, but you can get the response envelope, and access the headers through that. You can access the response envelope from the result of the asynchronous API call. This is an `gw.xml.ws.AsyncResponse` object. On this object, get the `ResponseEnvelope` property. For SOAP 1.2 envelopes, the type of that response is type `gw.xsd.w3c.soap12_envelope.Envelope`. For SOAP 1.1, the type is the same except with `"soap11"` instead of `"soap12"` in the name.

From that object, get the headers in the `Header` property. That property contains a list of XML objects that represent all the headers.

Server Override URL

To override the server URL, for example for a test-only configuration, set the `ServerOverrideUrl` property on the `WsdlConfig` object for your API reference. It takes a `java.net.URI` object for the URL.

For example:

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

service.Config.ServerOverrideUrl = new URI("http://testingserver/xx")
s

// call a method on the service
service.helloWorld()
```

Implementing Advanced Web Service Security with WSS4J

For security options beyond HTTP Basic authentication and optional SOAP header authentication, you can use an additional set of APIs to implement whatever additional security layers.

For example, you might want to add additional layers of encryption, digital signatures, or other types of authentication or security.

From the SOAP client side, the way to add advanced security layers to outgoing requests is to apply transformations of the stream of data for the request. You can transform the data stream incrementally as you process bytes in the stream. For example, you might implement encryption this way. Alternatively, some transformations might require getting all the bytes in the stream before you can begin to output any transformed bytes. Digital signatures would be an example of this approach. You may you use multiple types of transformations. Remember that the order of them is important. For example, an encryption layer followed by a digital signature is a different output stream of bytes than applying the digital signature and then the encryption.

Similarly, getting a response from a SOAP client request might require transformations to understand the response. If the external system added a digital signature and then encrypted the XML response, you need to first decrypt the response, and then validate the digital signature with your keystore.

The standard approach for implementing these additional security layers is the Java utility WSS4J, but you can use other utilities as needed. The WSS4J utility includes support for the WSS security standard.

Outbound Security

To add a transformation to your outgoing request, set the `RequestTransform` property on the `WsdlConfig` object for your API reference. The value of this property is a Gosu block that takes an input stream (`InputStream`) as an argument and returns another input stream. Your block can do anything it needs to do to transform the data.

Similarly, to transform the response, set the `ResponseTransform` property on the `WsdlConfig` object for your API reference.

The following simple example shows you could implement a transform of the byte stream. In this example, the transform is an XOR (exclusive OR) transformation on each byte. In this simple example, simply running the transformation again decodes the request.

The following code implements a service that applies the transform to any input stream. The code that actually implements the transform is as follows. This is a web service that you can use to test this request.

The class defines a static variable that contains a field called `_xorTransform` that does the transformation.

```
package gw.xml.ws

uses java.io.ByteArrayInputStream
uses gw.util.StreamUtil
uses java.io.InputStream

class WsiTransformService {

  // THE FOLLOWING DECLARES A GOSU BLOCK THAT IMPLEMENTS THE TRANSFORM
  public static var _xorTransform( is : InputStream ) : InputStream = \ is ->{
    var bytes = StreamUtil.getContent( is )
    for ( b in bytes index idx ) {
      bytes[ idx ] = ( b ^ 17 ) as byte // xor encryption
    }
    return new ByteArrayInputStream( bytes )
  }

}
```

The following code connects to the web service and applies this transform on outgoing requests and the reply.

```
package gw.xml.ws

uses gw.testharness.TestBase
uses gw.testharness.RunLevel
uses org.xml.sax.SAXParseException

@RunLevel( NONE )
class WsiTransform {

  function testTransform() {
    var ws = new wsi.MyService.WsiTransformTestService()
    ws.Config.RequestTransform = WsiTransformTestService._xorTransform
    ws.Config.ResponseTransform = WsiTransformTestService._xorTransform

    ws.add( 3, 5 ) // call some method, and the transform is automatic
  }
}
```

One-Way Methods

A typical WS-I method invocation has two parts: the SOAP request, and the SOAP response. Additionally, WS-I supports a concept called *one-way methods*. A one-way method is a method defined in the WSDL to provide no SOAP response at all. The transport layer (HTTP) may send a response back to the client, however, but it contains no SOAP response.

Gosu fully supports calling one-way methods. Your web service client code does not have to do anything special to handle one-way methods. Gosu handles them automatically if the WSDL specifies a method this way.

IMPORTANT Be careful not to confuse *one-way methods* with *asychronous methods*. For more information about asynchronous methods, see “Asynchronous Methods” on page 233.

Asynchronous Methods

Gosu supports optional asynchronous calls to web services. Gosu exposes alternate web service methods signatures on the service with the `async_` prefix. Gosu does not generate the additional method signature if the method is a one-way method. The asynchronous method variants return an `AsyncResponse` object. Use that object with a polling design pattern (check regularly whether it is done) the choose to get results later (synchronously in relation to the calling code).

See the introductory comments in “Consuming WS-I Web Service Overview” on page 225 for related information about the basic types of connections for a method.

IMPORTANT Be careful not to confuse *one-way methods* with *asychronous methods*. For more information about one-way methods, see “One-Way Methods” on page 233.

The `AsyncResponse` object contains the following properties and methods:

- `start` method - initiates the web service request but does not wait for the response
- `get` method - gets the results of the web service, waiting (blocking) until complete if necessary
- `RequestEnvelope` - a read-only property that contains the request XML
- `ResponseEnvelope` - a read-only property that contains the response XML, if the web service responded
- `RequestTransform` - a block (an in-line function) that Gosu calls to transform the request into another form. For example, this block might add encryption and then add a digital signature.
- `ResponseTransform` - a block (an in-line function) that Gosu calls to transform the request into another form. For example, this block might validate a digital signature and then decrypt the data.

The following is an example of calling the a synchronous version of a method contrasted to using the asynchronous variant of it.

```
var ws = new ws.weather.Weather()

// Call the REGULAR version of the method
var r = ws.GetCityWeatherByZIP("94114")
print( "The weather is " + r.Description )

// Call the **asynchronous** version of the same method -- note the "async_" prefix to the method
var a = ws.async_GetCityWeatherByZIP("94114")

// by default, the async request does NOT start automatically. You must start it with
// the start() method.
a.start()

print("the XML of the request for debugging... " + a.RequestEnvelope)
print("")

print ("in a real program, you would check the result possibly MUCH later...")
```

```
// get the result data of this asynchronous call, waiting if necessary
var laterResult = a.get()
print("asynchronous reply to our earlier request = " + laterResult.Description)

print("response XML = " + a.ResponseEnvelope.asUTFString())
```

chapter 20

Java and Gosu

You can write Gosu code that uses Java types. Gosu code can instantiate Java types, access properties of Java types and call methods of Java types.

If you are considering writing Java code for your Gosu to call, consider instead writing that code directly in Gosu. Remember that from Gosu, you can do everything Java can do, including directly call existing Java classes and Java libraries. You can even write Gosu code enhancements that add properties and methods to Java types, and the new members are accessible from all Gosu code.

This topic describes how to write and deploy Java code to work with Gosu, and how to call it from Gosu.

IMPORTANT This topic does **not** discuss differences between the syntax of Gosu and Java. See "Gosu Introduction" on page 13 and "Notable Differences Between Gosu and Java" on page 37.

This topic includes:

Overview of Calling Java from Gosu

Gosu can directly use Java types as if they were native Gosu types.

Gosu can do all of the following:

- instantiate Java types
- manipulate Java objects (and primitives) as native Gosu objects.
- get variables from Java types
- call methods on Java types.

  Note: For methods that look like getters and setters, Gosu exposes methods instead as properties. For more information, see "Java Get and Set Methods Convert to Gosu Properties" on page 237.

- add new methods to Java types using Gosu *enhancements*.
- add new properties to Java types using Gosu *enhancements*. (readable, writable, or read/write)
- create Gosu classes that extend Java classes
- create Gosu interfaces that extend Java interfaces
- use Java enumerations
- use Java annotations

All of these features work with built-in Java types as well as your own Java classes and libraries. You can write Java classes that any Gosu code can call.

IMPORTANT This topic does **not** focus on differences between the syntax of Gosu and Java. For that information, refer to the topic “Gosu Introduction” on page 13 and “Notable Differences Between Gosu and Java” on page 37.

Java Classes are First-Class Types

The most important thing to do know about Gosu’s Java compatibility is that Java classes are first-class types in Gosu. For example, standard Java classes and custom Java classes can be instantiated with the `new` keyword:

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

Many Java Classes are Core Classes for Gosu

Many core Gosu classes actually reference Java types. For example:

- the class `java.util.String` is the basic text object class for Gosu code.
- the basic collection types in Gosu simply reference the Java versions, such as `java.util.ArrayList`.

```
print(list.get(0))
```

Java Packages in Scope

Many Java packages are in scope and thus do not need fully-qualified class names or explicit “uses” statements. All types in the package `java.lang.*` are automatically in scope.

So, although you could use the code:

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

The code is easier to understand with the simpler code:

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

Static Members in Gosu

Gosu supports *static* members (variables, functions, and property declarations) on a type. A static member means that the member exists only on the type (which exists only once), not on *instances* of the type. You can access static members on Java types just as you would native Gosu types.

For Gosu code that accesses static members, you must qualify the class that declares the static member. For example, to use the `Math` class’s cosine function and its static reference to value `PI`:

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

Gosu does not have an equivalent of the *static import* feature of Java 1.5, which allows you to omit the characters “`Math.`”.

This is only a syntax difference for using static members from Gosu (independent of whether the type is implemented in Gosu or Java). If you are writing Gosu code that calls static members of Java types, this does not affect how you write your Java code.

Simple Java Example

The following is a simple example of calling a Java class.

In your Java IDE, create and compile the following Java class called `Echo`:

```
package gw.doc.examples;

public class Echo {

  public String EchoStrings (String a, String b) {
  String modifiedA = "First Arg was " + a + "\n";
  String modifiedB = "Second Arg was " + b + "\n";
  return modifiedA + modifiedB;
  }

  public void PrintStrings (String a, String b) {
  String modifiedA = "First Arg was " + a + "\n";
  String modifiedB = "Second Arg was " + b + "\n";
  System.out.print(modifiedA + modifiedB);

   }


}
```

Next, choose a directory for your `.gsp` program and copy the `Echo.class` file to a subdirectory that mirrors the package of the class. For example, create an empty Gosu program file at this path:

```
MyDocuments/Gosu/Echo/Echo.gsp
```

Because the package is `gw.doc.examples`, add your Java class file to this location:

```
MyDocuments/Gosu/Echo/gsrc/gw/doc/examples/Echo.class
```

Next, open the `Echo.gsp` file in the Gosu Editor.

In the `Echo.gsp` file, paste the following simple program:

```
classpath "gsrc"

var e = new gw.doc.examples.Echo()

e.PrintStrings("hello", "world")
```

If you run this Gosu code, it prints:

```
First Arg was hello
Second Arg was world
```

Similarly, you can pass data between Java and Gosu:

```
classpath "gsrc"

var e = new gw.doc.examples.Echo();

var fromJava = e.EchoStrings("hello", "world")

print (fromJava)
```

Java Get and Set Methods Convert to Gosu Properties

Gosu can call methods on Java types. For methods on Java types that look like getters and setters, Gosu exposes methods instead as properties. Gosu uses the following rules for methods on Java types:

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

- Similar to the rules for `get`, the method name starts with `is` and takes no arguments and returns a Boolean value, Gosu exposes this as a property accessor (a *getter*). The property name matches the original method but without the prefix `is`. For example, suppose the Java method signature is `isVisible()`. Gosu exposes this a `property get` function for the property named `Visible`.

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

For example, create and compile this Java class:

```
package gw.doc.examples;

public class Circle {
  public static final double PI = Math.PI;
  private double _radius;

  //Constructor #1 - no arguments
  public Circle() {
  }

  //Constructor #2
  public Circle( int dRadius ) {
  _radius = dRadius;
  }

  // from Java these are METHODS that begin with get, set, is
  // from Gosu these are PROPERTY accessors

  public double getRadius() {
    System.out.print("running Java METHOD getRadius() \n");
  return _radius;
  }
  public void   setRadius(double dRadius) {
   System.out.print("running Java METHOD setRadius() \n");
  _radius = dRadius;
  }
  public double getArea() {
   System.out.print("running Java METHOD getArea() \n");
  return PI * getRadius() * getRadius();
  }
  public double getCircumference() {
   System.out.print("running Java METHOD getCircumference() \n");
  return 2 * PI * getRadius();
  }
  public boolean isRound() {
   System.out.print("running Java METHOD isRound() \n");
  return(true);
  }


  // ** the following methods stay as methods, not properties! **

  //  For GET/IS, the method must take 0 args and return a value
  public void isMethod1 () {
  System.out.print("running Java METHOD isMethod1() \n");
  }
  public double getMethod2 (double a, double b) {
  System.out.print("running Java METHOD isMethod2() \n");
    return 1;
  }

  //  For SET, the method must take 1 args and return void
  public void setMethod3 () {
   System.out.print("running Java METHOD setMethod3() \n");
  }
  public double setMethod4 (double a, double b) {
   System.out.print("running Java METHOD setMethod4() \n");
   return 1;
  }

}
```

The following Gosu code uses this Java class. Note which Java methods become property accessors and which ones do not.

```
// instatiate the class with the constructor that takes an argument
var c = new gw.doc.examples.Circle(10)
```

```
// Use natural property syntax to SET GOSU PROPERTIES. In Java, this was a method.
c.Radius = 10

// Use natural property syntax to GET GOSU PROPERTIES
print("Radius " + c.Radius)
print("Area " + c.Area)
print("Round " + c.Round) // boolean true  coerces to String "true"
print("Circumference " + c.Circumference)

// the following would be syntax errors if you uncomment. They are not writable (no setter method)
// c.Area = 3
// c.Circumference = 4
// c.Round = false

// These Java methods do not convert to properties (wrong number of arguments or wrong type)
c.isMethod1()
var temp2 = c.getMethod2(1,2)
c.setMethod3()
var temp4 = c.setMethod4(8,9)
```

This Gosu code outputs the following:

```
running Java METHOD setRadius()
running Java METHOD getRadius()
Radius 10
running Java METHOD getArea()
running Java METHOD getRadius()
running Java METHOD getRadius()
Area 314.1592653589793
running Java METHOD isRound()
Round true
running Java METHOD getCircumference()
running Java METHOD getRadius()
Circumference 62.83185307179586
running Java METHOD isMethod1()
running Java METHOD isMethod2()
running Java METHOD setMethod3()
running Java METHOD setMethod4()
```

Interfaces

Gosu classes can directly implement Java interfaces.

Gosu interfaces can directly extend Java interfaces.

Enumerations

Gosu can directly use Java enumerations.

Annotations

Gosu can directly use Java annotations.

Java Primitives

Gosu supports the following primitive types: `int`, `char`, `byte`, `short`, `long`, `float`, `double`, `boolean`, and the special value that means an empty object value: `null`. This is the full set that Java supports, and the Gosu versions are fully compatible with the Java primitives, in both directions.

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

In both Gosu and Java, the language primitive types like `int` and `boolean` work differently from objects (descendents of the root `Object` class). For example:

- you can add objects to a collection, but not primitives
- variables typed to an object type can have the value `null`, but this is not true for primitives

The Java classes `java.lang.Boolean` and `java.lang.Integer` are `Object` types and can freely be used within Gosu code because of Gosu's special relationship to the Java language. These wrapper objects are referred to as *boxed types* as opposed to the primitive values as *unboxed types*.

Gosu can automatically convert values from unboxed to Java-based boxed types as required by the specific API or return value, a feature that is called *autoboxing*. Similarly, Gosu can automatically convert values from boxed to boxed types, a feature that is called *unboxing*.

In most cases, you do not need to worry about differences between boxed and unboxed types because Gosu automatically converts values as required. For example, Gosu implicitly converts between the native language primitive type called `boolean` and the Java-based object class `Boolean` (`java.util.Boolean`). In cases you want explicit coercion, simply use the "`as ...NEWTYPE`" syntax, such as "`myIntValue as Integer`".

If your code implicitly converts a variable's value from a boxed type to a unboxed type, if the value is `null`, Gosu standard value type conversion rules apply. For example:

```
var bBoxed : Boolean
var bUnboxed : boolean

bBoxed = null      // bBoxed can genuinely be NULL
bUnboxed = bBoxed  // bUnboxed can't be null, so is converted to FALSE!
```

For more information, see "Type Object Properties" on page 258.

Deploying Your Java Classes

Detailed Java Class Deployment Checklist Using Command Line Tool

If you are using Gosu as a self-contained command-line tool, you can deploy Java classes that your Gosu can call. You can put your class files and JAR files wherever you want. Your Java program can contain a directive that identifies one or more class path directories to look for class files and JAR files. The classpath can be a relative path or an absolute path. For details, see "The Structure of a Gosu Program" on page 59.

Java Class Deployment For IntelliJ IDEA IDE

Refer to the IntelliJ documentation for details about Java file management in IntelliJ IDEA.

Java Class Loading, Delegation, and Package Naming

Java Class Loading Rules

If loading custom Java code into Gosu or if accessing Java classes from Java code, the Java virtual machine must locate the class file with a *class loader*. Class loaders use the fully-qualified package name of the Java class to determine how to access the class.

Gosu follows the rules in the following list to load Java classes, choosing the first rule that matches and then skipping the rules listed after it:

1. **General delegation classes.**

   The following classes *delegate load*:

   - `javax.*` - Java extension classes
   - `org.xml.sax.*` - SAX 1 & 2 classes
   - `org.w3c.dom.*` - DOM 1 & 2 classes
   - `org.apache.xerces.*` - Xerces 1 & 2 classes

- `org.apache.xalan.*` - Xalan classes
- `org.apache.commons.logging.*` - Logging classes used by WebSphere

2. All your classes.

If the package does **not** begin with `com.guidewire.*`, then *load locally.*

WARNING Java classes you deploy must **never** have a fully-qualified package name that starts with `"com.guidewire."` because that interferes with class loading behavior.

---

3. Internal classes.

If the class is an internal class, then the class *delegate loads*.

WARNING Java code you deploy must **never** access any internal classes other than supported classes and documented APIs. Using internal classes is dangerous and unsupported.

---

chapter 21

Gosu Templates

Gosu includes a native template system. Templates are text with embedded Gosu code within a larger block of text. The embedded Gosu code optionally can calculate a value and export the result as text in the location the code appears in the template text.

This topic includes:

Template Overview

Templates are text with embedded Gosu code within a larger block of text. The embedded Gosu code optionally can calculate a value and export the result as text in the location the code appears in the template text. There are two mechanisms to use Gosu templates:

- **Template syntax inside text literals.** Inside your Gosu code, use template syntax for an inline `String` literal values with embedded Gosu expressions. Gosu template syntax combines static text that you provide with dynamic Gosu code that executes at run time and returns a result. Gosu uses the result of the Gosu expression to output the dynamic output at run time as a `String` value.
- **Separate template files.** Define Gosu templates as separate files that you can execute from other code to perform actions and generate output. If you use separate template files, there are additional features you can use such as passing custom parameters to your template. For more details, see "Using Template Files" on page 245.

The simplest way to use to templates is to embed Gosu expressions that evaluate at run time and generate text in the place of the embedded Gosu expressions.

Template Expressions

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

```
${ EXPRESSION }
```

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

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

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

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

If you print this variable, Gosu outputs:

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

Gosu runs your template expression at run time. The expression can include variables or dynamic calculations that return a value:

```
var s1 = "One plus one equals ${ myVariable }."
var s2 = "The total is ${ myVariable.calculateMyTotal() }."
```

At compile time, Gosu uses the built-in type checking system to ensure the embedded expression is valid and type safe.

If the expression does not return a value of type `String`, Gosu attempts to coerce the result to the type `String`.

Alternate Template Expression Syntax <%= … %>

The syntax `${ EXPRESSION }` is the preferred style for template expressions.

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

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

Any surrounding text exports to the output directly.

When to Escape Special Characters for Templates

Gosu templates use standard characters in the template to indicate the beginning of a special block of Gosu code or other template structures. In some cases, to avoid ambiguity for the Gosu parser you must specially escape special characters.

For Non-Template-Tag Use, Escape ${ or <%

Gosu templates use the following two-character sequences to begin a template expression

- `${`
- `<%`

With a `String` literal in your code, if you want to use these to indicate template tags, do not need to escape these special characters.

If you want either of those two special two-character sequences actually in your `String` (not as a template tag), escape the first character of that sequence. To escape a character, add a backslash character immediately before it. For example:

- To define a variable containing the non-template text `"Hello${There"`:
  ```
  var s = "Hello\${There"
  ```
- To define a variable containing the non-template text `"Hello<%There"`:
  ```
  var s = "Hello\<%There"
  ```

If you use the initial character on its own (the next character would not indicate a special tag), you do not need to escape it. For example:

- To define a variable containing the non-template text "Hello$There", simply use:

```
var s = "Hello$There"
```

- To define a variable containing the non-template text "Hello<There", simply use:

```
var s = "Hello<There"
```

Within Template Tag Blocks, Use Standard Gosu Escaping Rules

In typical use, if you defined a String, you must escape it with the syntax \" to avoid ambiguity about whether you were ending the String. For example:

```
var quotedString = "\"This has double quotes around it\", is that correct?"
```

This creates a String with the following value, including quote signs:

```
"This has double quotes around it", is that correct?
```

However, if you use a template, this rule does not apply between your template-specific open and closing tags that contain Gosu code. Instead, use standard Gosu syntax for the code between those open and closing tags.

In other words, the following two lines are valid Gosu code:

```
var s = "${ "1" }"
var s = "${ "1" } \"and\" ${ "1" }"
```

Note that the first character within the template's Gosu block is an unescaped quote sign.

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

```
var s = "${ \"1\" }"
```

In this invalid case, the first character within the template's Gosu block is an escaped quote sign.

In the rare case that your Gosu code requires creating a String literal containing a quote character, remember that the standard Gosu syntax rules apply. This means that you will need to escape any double quote signs that are within the String literal. For example, the following is valid Gosu:

```
var quotedString = "${ "\"This has double quotes around it\", is that correct?" }"
```

Note that the first character within the template's Gosu block is an unescaped quote sign. This template generates a String with the value:

```
"This has double quotes around it", is that correct?
```

IMPORTANT Be careful with how you escape double quote characters within your embedded Gosu code or other special template blocks.

Using Template Files

Instead of defining your templates in inline text, you can store a Gosu template as a separate file. Template files support all the features that inline templates support, as described in "Template Overview" on page 243. In addition, with template files you get additional advantages and features:

- **Separate your template definition from code that uses the template.** For example, define a template that generates a report or a notification email. You can then call this template from many places but define the template only once.
- **Encapsulate your template definition for better change control.** By defining the template in a separate file, your teams can edit and track template changes over time separate from code that uses the template.
- **Run Gosu statements (and return no value) using scriptlet syntax.** You can define one or more Gosu statements as a *statement list* embedded in the template. Contrast this with the template expression syntax described in "Template Overview" on page 243, which require Gosu *expressions* rather than Gosu *statements*. The result of scriptlet tags generate no output. For more information, see "Template Scriptlet Tags" on page 246.

- **Define template parameters.** Template files can define parameters that you pass to the template at run time. For more information, see “Template Parameters” on page 247.
- **Extend a template from a class to simplify static method calls.** If you call static methods on one main class in your template, you can simplify your template code using the `extends` feature. For more information, see “Extending a Template From a Class” on page 248.

Creating and Running a Template File

Gosu template files have the extension `.gst`. Create template files within the package hierarchy in the file system just as you create Gosu classes. Choose the package hierarchy carefully because you use this package name to access and run your template.

In your template file, include the template body with no surrounding quote marks. The following is a simple template:

```
One plus one equals ${ 1 + 1 }.
```

To create a new template, add a template file (ending with `.gst`) in the class hierarchy. Add it within the directory whose path matches the package in which you want this template to appear. For example, your Gosu program can have a

The template is a first-class object in the Gosu type system within its package namespace. To run a template, get a reference to your template and call the `renderToString` method of the template.

For example, suppose you create a template file `NotifyAdminTemplate.gst` within the package `mycompany.templates`. Your fully-qualified name of the template is `mycompany.templates.NotifyAdminTemplate`.

Use the following code to render (run) your template:

```
var x = mycompany.templates.NotifyAdminTemplate.renderToString()
```

The variable `x` contains the `String` output of your template.

If you want to pass template parameters to your template, add additional parameters as arguments to the `renderToString` method. See “Template Parameters” on page 247 for details.

Output to a Writer

The `renderToString` method outputs the template results to a `String` value. Optionally, you can render the template directly to a Java writer object. Your writer must be an instance of `java.io.Writer`. To output to the writer, get a reference to the template and call its `render` method, passing the writer as an argument to the method.

For example, suppose you create a template file `NotifyAdminTemplate.gst` within the package `mycompany.templates`. If your variable `myWriter` contains an instance of `java.io.Writer`, the following Gosu statement renders the template to the writer:

```
mycompany.templates.NotifyAdminTemplate.render(myWriter)
```

If you use template parameters in your template, add your additional parameters after the writer argument. See “Template Parameters” on page 247 for details.

Template Scriptlet Tags

Text enclosed with the text `<%` and `%>` evaluate at run time in the order the parser encounters the text but generates nothing as output based on the result. These are called *scriptlet tags*. It is important to note that this type of tag has no equals sign in the opening tag. All plain text between scriptlet tags generate to the output within the scope and the logic of the scriptlet code.

The following simple template uses a scriptlet tag to run code to assign a variable and uses the result later:

```
<% var MyCalc = 1 + 2 %>One plus two is ${ MyCalc }
```

This prints the following:

```
One plus two is 3
```

It is important to note that the result of the scriptlet tag at the beginning of the template does **not** generate anything to the output. The value `3` exports to the result because later expression surrounded with the *expression* delimiters `${` and `}` instead of the scriptlet delimiters `<%` and `%>`.

The scope of the Gosu continues **across** scriptlet tags. Use this feature to write advanced logic that uses Gosu code that you spread across multiple scriptlet tags. For example, the following template code outputs “x is 5” if the variable x has the value 5, otherwise outputs `"x is not 5"`:

```
<% if (x == 5) { %>
x is 5
<% } else { %>
x is not 5
<% } %>
```

Notice that the `if` statement actually controls the flow of execution of later elements in the template. This feature allows you to control the export of static text in the template as well as template expressions.

Scriptlet tags are particularly useful when used with template parameters because you can define conditional logic as shown in the previous example. See “Template Parameters” on page 247 for details.

Use this syntax to iterate across lists, arrays, and other iterable objects. You can combine this syntax with the expression syntax to generate output from the inner part of your loop. Remember that the scriptlet syntax does not itself support generating output text.

For example, suppose you set a variable called `MyList` that contains a `List` of objects with a `Name` property. The following template iterates across the list:

```
<% for (var obj in MyList) {
  var theName = obj.Name %>
  Name: ${ theName }
<% } %>
```

This might generate output such as:

```
Name: John Smith
Name: Wendy Wheathers
Name: Alice Applegate
```

This example also shows a common design pattern for templates that need to combine complex logic in scriptlet syntax with generated text into the template within a loop:

1. Begin a template scriptlet (starting it with `<%`) to begin your loop.
2. Before ending the scriptlet, set up a variable with your data to export
3. End the scriptlet (closing it with `%>`).
4. Optionally, generate some static text
5. Insert a template expression to export your variable, surrounding a Gosu expression with `${` and `}` tags.
6. Add another template scriptlet (with `<%` and `%>`) to contain code that closes your loop. Remember that scriptlets share scope across all scriptlets in that file, so you can reference other variables or close loops or other Gosu structural elements.

IMPORTANT There is no supported API to generate template output from within a template scriptlet. Instead, design your template to combine template scriptlets and template expressions using the code pattern in this topic.

The scriptlet tags are available in template files, but not within `String` literals using template syntax.

Template Parameters

You can pass parameters of any type to your self-contained Gosu template files.

To support parameters in a template

1. Create a template file as described earlier in this topic.
2. At the top of the template, create a parameter definition with the following syntax:
   ```
   <%@ params(ARGLIST) %>
   ```
   In this case, *ARGLIST* is an argument list just as with a standard Gosu function. For example, the following argument list includes a `String` argument and a `boolean` argument:
   ```
   <%@ params(x : String, y : boolean) %>
   ```
3. Later in the template, use template tags that use the values of those parameters. You can use both the template expression syntax (`${` and `}`) and template scriptlet syntax (`<%` and `%>`). Remember that the expression syntax always returns a result and generates additional text. In contrast, the scriptlet syntax only executes Gosu statements.
4. To run the template, add your additional parameters to the call to the `renderToString` method or after the writer parameter to the `render` methods.

For example, suppose you create a template file `NotifyAdminTemplate.gst` within the package `mycompany.templates`. Edit the file to contain the following contents:

```
<%@ params(personName : String, contactHR: boolean) %>
The person ${ personName } must update their contact information in the company directory.

<% if (contactHR) { %>
Call the human resources department immediately.
<% } %>
```

In this example, the `if` statement (including its trailing curly brace) is within scriptlet tags. The `if` statement uses the parameter value at run time to conditionally run elements that appear later in the template. This template exports the warning to call the human resources department **only** if the `contactHR` parameter is `true`. Use if statements and other control statements to control the export of static text in the template as well as template expressions.

Run your template with the following code:

```
var x : String = mycompany.templates.NotifyAdminTemplate.renderToString("hello", true)
```

If you want to export to a character writer, use code like the following:

```
var x : String = mycompany.templates.NotifyAdminTemplate.render(myWriter, "hello", true)
```

You can use template parameters in template files, but not within `String` literals that use template syntax.

Extending a Template From a Class

Gosu provides a special syntax to simplify calling static methods on a class of your choosing. The metaphor for this template shortcut is that your template can *extend* from a type that you define. Technically, templates are not instantiated as objects. However, your template can call **static** methods on the specified class without fully-qualifying the class. Static methods are methods defined directly on a class, rather than on instances of the class. For more information, see "Modifiers" on page 135.

To use this feature, at the top of the template file, add a line with the following syntax:

```
<%@ extends CLASSNAME %>
```

CLASSNAME must be a fully-qualified class name. You cannot use a package name or hierarchy.

For example, suppose your template wants to clean up the email address with the `sanitizeEmailAddress` static method on the class `gw.api.email.EmailTemplate`. The following template takes one argument that is an email address:

```
<%@ params(address : String) %>
<%@ extends gw.api.email.EmailTemplate %>
Hello! The email address is ${sanitizeEmailAddress(address)}
```

Notice that the class name does **not** appear immediately before the call to the static method.

You can use the `extends` syntax in template files, but not within `String` literals that use template syntax.

Template Comments

You can add comments within your template. Template comments do not affect template output.

The syntax of a template comments is the following:

```
<%-- your comment here --%>
```

For example:

```
My name is <%-- this is a comment --%>John.
```

If you render this template file, it outputs:

```
My name is John.
```

You can use template comments in template files, but not within `String` literals that use template syntax.

Template Export Formats

Because HTML and XML are text-based formats, there is no fundamental difference between designing a template for HTML or XML export compared to a plain text file. The only difference is that the text file must conform to HTML and XML specifications.

HTML results must be a well-formed HTML, ensuring that all properties contain no characters that might invalidate the HTML specification, such as unescaped "`<`" or "`&`" characters. This is particularly relevant for especially user-entered text such as descriptions and notes.

Systems that process XML typically are **very strict** about syntax and well-formedness. Be careful not to generate text that might invalidate the XML or confuse the recipient. For instance, beware of unescaped "`<`" or "`&`" characters in a notes field. If possible, you could export data within an XML `<CDATA>` tag, which allows more types of characters and character strings without problems of unescaped characters.

chapter 22

Type System

Gosu provides several ways to gather information about an object or other type. This ability for a programming language to query an object from the outside for this information is referred to as *reflection*. Use this information for debugging or to change program behavior based on information gathered at run time.

This topic includes:

Basic Type Coercion

Gosu uses the "`expression as TYPE`" construction to cast an expression to a specific type. This process is also called *coercion*.

Syntax

```
expression as TYPE
```

The expression must be compatible with the type. The following table lists the results of casting a simple numeric expression into one of the Gosu-supported data types. If you try to cast an expression to an inappropriate type, Gosu throws an exception.

Expression	Data type	Result or error
`(5 * 4) as Array`	*n/a*	`Type mismatch or possible invalid operator.` `java.lang.Object[] is not compatible with java.lang.Double`
`(5 * 4) as Boolean`	`Boolean`	`true`
`(5 * 4) as DateTime`	`DateTime`	`1969-12-31` (default value)

Expression	Data type	Result or error
`(5 * 4) as String`	`String`	`20`
`(5 * 4) as Type`	*n/a*	`Type mismatch or possible invalid operator.` `MetaType:java.lang.Object is not compatible with java.lang.Double`

Why Use Coercion?

Gosu requires all variables to have types at compile type. All method calls and properties on objects must be correct with the compile time type.

If an object has a compile-time type that is higher in the type hierarchy (it is a supertype) than you need, coerce it to the appropriate specific type. This is required before accessing properties or methods on the object that are defined on a more specific type.

For example, when getting items from a list or array, the compile time type might be a supertype of the type you know that it is. For example, the compile time type is an `Object` but you know that it is always a more specific type due to your application logic. You must cast the item to the desired type before accessing properties and methods associated with the subtype you expect.

The following example coerces an `Object` to the type `MyClass` so the code can call `MyMethod` method. This example assumes `MyMethod` is a method on the class `MyClass`:

```
var objarray : Object[] = MyUtilities.GetMyObjectArray()

var o = objarray[0] // type of this variable is Object

var myresult = (o as MyClass).MyMethod()
```

Gosu provides automatic downcasting to simplify your code in `if` statements and similar structures. For more information, see “Automatic Downcasting for ‘typeis’ and ‘typeof’” on page 254.

For related information, see “Basic Type Checking” on page 252 and “Using Reflection” on page 256.

Basic Type Checking

Gosu uses the `typeis` operator to compare an expression’s type with a specified type. The result is always `Boolean`. A `typeis` expression cannot be fully determined at compile time. For example, at run time the expression may evaluate to a more specific subtype than the variable is declared as.

Typeis Syntax

```
OBJECT typeis TYPE
```

Typeis Examples

Expression	Result
`42 typeis Number`	`true`
`"auto" typeis String`	`true`
`person typeis Person`	`true`
`person typeis Company`	`false`

Similarly, you can use the `typeof` *`object`* operator to test against a specific type.

Typeof Syntax

```
typeof expression
```

Typeof Examples

Expression	Result
`typeof 42`	`Number`
`typeof "auto"`	`String`
`typeof (4 + 5)`	`Number`

In real-world code, typically you need to check an object against a type **or** its subtypes, not just a single type. In such cases, it is better to use `typeis` instead of `typeof`. The `typeof` keyword returns the exact type. If you test this value with simple equality with another type, it returns `false` if the object is a subtype.

For example, the following expression returns `true`:

```
"hello" typeis Object
```

In contrast, the following expression returns `false` because `String` is a subtype of `Object` but is a different type:

```
typeof "hello" == Object
```

If you want information from the type itself, you can access a type by name (typing its *type literal*) or use the `typeof` operator to get an object's type. For example:

```
var s = "hello"
var t = typeof s
```

In this example, the type of `s` is `String`, so the value of the `t` variable is now `String`.

Static Type ('statictypeof')

To get the compile-time type of an object and use it programmatically, use the `statictypeof` keyword. The result of an `statictypeof` expression does not vary at run time. Contrast this with the `typeof` keyword, which performs a run-time check of the object.

The following example illustrates this difference:

```
var i : Object = "hello"
print(typeof i)
print(statictypeof i)
```

This example prints the output:

```
java.lang.String
java.lang.Object
```

The variable is declared as `Object`. However, at run time it contains an object whose type is `String`, which is a subtype of `String`.

The following example also illustrates how this difference can affect `null` values and unexpected conditions:

```
var i : Boolean;
i = null;
print(typeof i)
print(statictypeof i)
```

This prints the output:

```
void
java.lang.Boolean
```

At run time, the value of `i` is `null`, so its type is `void`. However, the compile-time type of this variable is `Boolean`.

Is Assignable From

For advanced manipulation of type objects, including the method called `isAssignableFrom` that exists on types, see "Using Reflection" on page 256.

Even Types Have Types

All objects have types. This even applies to types (such as the type called `String`). The expression `typeof String` evaluates to a parameterized version of the type `Type`. Specifically:

```
Type<java.lang.String>
```

For advanced manipulation of the `Type` object, see “Using Reflection” on page 256.

Automatic Downcasting for ‘typeis’ and ‘typeof’

To improve the readability of your Gosu code, Gosu automatically downcasts after a `typeis` expression if the type is a subtype of the original type. This is particularly valuable for `if` statements and similar Gosu structures. Within the Gosu code bounded by the `if` statement, you do not need to do casting (`as` *`TYPE`* expressions) to that subtype. Because Gosu confirms that the object has the more specific subtype, Gosu implicitly considers that variable’s type to be the **subtype**, at least within that block of code.

For example, a common pattern for this feature looks like the following:

```
var VARIABLE_NAME : TYPE_NAME

if (VARIABLE_NAME typeis SUBTYPE_NAME) {

  // use the VARIABLE_NAME as SUBTYPE_NAME without casting
  // This assumes SUBTYPE_NAME is a subtype of TYPE_NAME
}
```

For example, the following example shows a variable declared as an `Object`, but downcasted to `String` within the `if` statement.

Because of downcasting, the following code is valid:

```
var x : Object = "nice"
var strlen = 0

if( x typeis String ) {
  strlen = x.length
}
```

It is important to note that `length` is a property on `String`, not `Object`. The downcasting from `Object` to `String` means that you do not need an additional casting around the variable `x`. In other words, the following code is equivalent but has an **unnecessary** cast:

```
var x : Object = "nice"
var strlen = 0

if( x typeis String ) {
  strlen = (x as String).length // "length" is a property on String, not Object
}
```

Do not write Gosu code with unnecessary casts. Use automatic downcasting to write easy-to-read and concise Gosu code.

The automatic downcasting happens for the following types of statements;

- `if` statements. For more information, see “If() … Else() Statements” on page 102.
- `switch` statements. For more information, see “Switch() Statements” on page 105. For example:

```
uses java.util.Date

var x : Object = "neat"
switch( typeof( x ) ){
  case String :
    print( x.charAt( 0 ) ) // without automatic downcasting, this method call fails without casting
    break
  case Date :
    print( x.Time ) // without automatic downcasting, this property access fails without casting
    break
}
```

- ternary conditional expression, such as “`x typeis String ? x.length : 0`”. Downcasting only happens in the part of the expression that corresponds to it being true (the first part). For more information, see “Conditional Ternary Expressions” on page 93.

This automatic downcasting works when the item to the left of the `typeis` keyword is a symbol, but not on other expressions.

There are a several situations that cancel the `typeis` inference:

- Reaching the end of the extent of the scope for which inference is appropriate. In other words:
  - The end of an `if` statement
  - The end of a `switch` statement
  - The end of a ternary conditional expression in its `true` clause
- Assigning any value to the symbol (the variable) you checked with `typeis` or `typeof`. This applies to `if` and `switch` statements.
- An `or` keyword in a logical expression
- The end of an expression negated with the `not` keyword
- In a `switch` statement, a `case` section does not use automatic downcasting if the previous case section is unterminated by a `break` statement. For example, the following Gosu code us valid and both `case` sections using automatic downcasting:

```
uses java.util.Date

var x : Object = "neat"
switch( typeof( x ) ){
  case String :
    print( x.charAt( 0 ) ) // without automatic downcasting, this method call fails without casting
    break
  case Date :
    print( x.Time ) // without automatic downcasting, this property access fails without casting
    break
}
```

  However, Gosu allows you to remove the first `break` statement. Removing a break statement allows the execution to fall through to the next `case` section. In other words, if the type is `String`, Gosu runs the `print` statement in the `String` case section. Next, Gosu run statements in the next `case` section also. This does not change the type system behavior of the section whose `break` statement is now gone (the first section). However, there is no downcasting for the following `case` section since two different cases share that series of Gosu statements. The compile time type of the switched object reverts to the compile-time type of that variable at the beginning of the `switch` statement.

  For example, the following code has a compile error because it relies on downcasting.

```
uses java.util.Date

var x : Object = "neat"
switch( typeof( x ) ){
  case String :
    print( x.charAt( 0 ) ) // without automatic downcasting, this method call fails without casting
  case Date :
    print( x.Time ) // COMPILE ERROR. The compile time type reverts to Object (no Time property!)
    break
}
```

  To work around this problem, remember that the compile time type of the switched object reverts to whatever the compile-time type is before the `switch` statement. Simply cast the variable with the `as` keyword before accessing type-specific methods or properties. For example:

```
uses java.util.Date

var x : Object = "neat"
switch( typeof( x ) ){
  case String :
    print( x.charAt( 0 ) ) // without automatic downcasting, this method call fails without casting
  case Date :
    print( (x as Date).Time ) // this is now valid Gosu code
    break
}
```

Using Reflection

Once you know what type something is, you can use reflection to learn about the type. Although each Type object itself has properties and methods on it, the most interesting properties and methods are on the *type*.TypeInfo object. For example, you can get a type's complete set of properties and methods at run time by getting the TypeInfo object.

> **WARNING** In general, avoid using reflection to get properties or call methods. In almost all cases, you can write Gosu code to avoid reflection. Using reflection dramatically limits how Gosu can alert you to serious problems at compile time. In general, it is better to detect errors at compile time rather than unexpected behavior at run time. Only use reflection if there is no other way to do what you need.

The following example shows two different approaches for getting the Name property from a type:

```
print(Integer.Name)     // directly from a Type
print((typeof 29).Name) // getting the Type of something
```

This prints:

```
java.lang.Integer
int
```

Get Properties Using Reflection

The *type*.TypeInfo object includes a property called properties, that contains a list of type properties.

Each item in that list include metadata properties such as for the name (Name) and a short description (ShortDescription).

For example, paste the following code into the Gosu Tester:

```
var object = "this is a string"
var s = ""
var props =  (typeof object).TypeInfo.Properties

for (m in props) {
  s = s + m.Name + "  "
}

print(s)
```

This code prints something similar to the following:

```
Class  itype  Bytes  Empty  CASE_INSENSITIVE_ORDER  length  size  HasContent
NotBlank  Alpha  AlphaSpace  Alphanumeric  AlphanumericSpace  Numeric  NumericSpace  Whitespace
```

You can also call properties using reflection using the square bracket syntax, similar to using arrays. For example, paste the following code into the Gosu Tester:

```
// get the CURRENT time
var s = new DateTime()

// createa  String containing a property name
var propName = "hour"

// get a property name using reflection
print(s[propName])
```

If the time is currently 5 PM, this code prints:

```
5
```

Get Methods Using Reflection

Paste the following code into the Gosu Tester:

```
var object = "this is a string"
var s = ""
var methods =  (typeof object).TypeInfo.Methods

for (m in methods) {
  s = s + m.Name + "  "
}
```

```
print(s)
```

This code prints code that looks like this (truncated for clarity):

```
wait()  wait( long, int )  wait( long )  hashCode()  getClass()  equals( java.lang.Object )
toString()  notify()  notifyAll()  @itype()  compareTo( java.lang.String )  charAt( int )
length()  subSequence( int, int )  indexOf( java.lang.String, int )  indexOf( java.lang.String )
indexOf( int )  indexOf( int, int )  codePointAt( int )  codePointBefore( int )
```

You can also get information about individual methods. You can even call methods by name (given a `String` for the method name) and pass a list of parameters as `object` values. You can call a method using the method's `CallHandler` property, which contains a `handleCall` method.

The following example gets a method by name and then calls it. This example uses the `String` class and its `compareTo` method, which returns 0, 1, or -1. Paste the following code into the Gosu Tester

```
var mm = String.TypeInfo.Methods
var myMethodName = "compareTo"

// find a specific method by name using "collections" and "blocks" features...
var m = mm.findFirst( \ i -> i.Name == myMethodName )

print("Name is " + m.Name)
print("Number of parameters is " + m.Parameters.length)
print("Name of first parameter is " + m.Parameters[0].DisplayName)
print("Type of first parameter is " + m.Parameters[0].IntrinsicType)

// set up an object whose method to call. in this case, use a String
var obj = "a"
var comparisonString = "b"

// call the method using reflection! ** note: this returns -1 because "a" comes before "b"
print(m.CallHandler.handleCall( obj, { comparisonString } ))

// in this example, this was equivalent to the code:
print(obj.compareTo(comparisonString))
```

This code prints:

```
Name is compareTo
Number of parameters is 1
Name of first parameter is String
Type of first parameter is java.lang.String
-1
-1
```

Compare Types Using Reflection

You can compare the type of two objects in several ways.

You can use the equality (==) operator to test types. However, the equality operator is almost always inappropriate because it returns `true` only for exact type matches. It returns `false` if one type is a subtype of the other or if the types are in different packages.

Instead, use the *`type`*`.isAssignableFrom(`*`otherType`*`)` method to determine whether the types are compatible for assignment. This method considers the possibility of subtypes (such as subclasses) in a way that the equality operator does not. The method determines if the type argument is either the same as, or a superclass of (or super-interface of) the type.

The *`sourceType`*`.isAssignableFrom(`*`destinationType`*`)` method looks only at the supertypes of the source type. Although Gosu statements can assign a value of one **unrelated** type to another using coercion, the `isAssignableFrom` method always returns `false` if coercion of the data would be necessary. For example, Gosu can convert `boolean` to `String` or from `String` to `boolean` using coercion, but `isAssignableFrom` method returns `false` for those cases.

Gosu provides a variant of this functionality with the Gosu `typeis` operator. Whereas *`type`*`.isAssignableFrom(...)` operates between a type and another type, the `typeis` operates between an object and a type.

Paste the following code into the Gosu Tester:

For example, paste the following code into the Gosu Tester:

```
var s : String = "hello"
var b : Boolean = true

print("Typeof s: " + (typeof s).Name)
print("Boolean assignable from String : " + (typeof s).isAssignableFrom((typeof b)))
print("true typeis String: " + (b typeis String))
print("Object assignable from String: " + (Object).isAssignableFrom( String ))
print("Compare a string to object using typeis: " + (s typeis Object))

// Using == to compare types is a bad approach if you want to check for valid subtypes...
print("Compare a string to object using == : " + ((typeof s) == Object))
```

This code prints:

```
Typeof s: java.lang.String
Boolean assignable from String : false
true typeis String: false
Object assignable from String: true
Compare a string to object using typeis: true
Compare a string to object using == : false
```

Type Object Properties

The `Type` type is a metatype, which means that it is the type of all types. There are various methods and properties that appear directly on the type `Type` and all are supported.

The `Type` type includes the following important properties:

Property	Description
`Name`	The human-readable name of this type.
`TypeInfo`	Properties and methods of this type. See “Basic Type Checking” on page 252 for more information and examples that use this `TypeInfo` object.
`SuperType`	The supertype of this type, or `null` if there is no supertype.
`IsAbstract`	If the type is abstract, returns `true`. See “Modifiers” on page 135.
`IsArray`	If the type is an array, returns `true`.
`IsFinal`	If the type is final, returns `true`. See “Modifiers” on page 135.
`IsGeneric`	If the type is generic, returns `true`. See “Gosu Generics” on page 173.
`IsInterface`	If the type is an interface, returns `true`. See “Interfaces” on page 147.
`IsParameterized`	If the type is parameterized, returns `true`. See “Gosu Generics” on page 173.
`IsPrimitive`	If the type is primitive, returns `true`.

For more information about the `isAssignableFrom` method on the `Type` object, refer to the previous section.

Working with Primitive Types

In Gosu, primitive types such as `int` and `boolean` exist primarily for compatibility with the Java language. Gosu uses these Java primitive types to support extending Java classes and implementing Java interfaces. From a Gosu language perspective, primitives are different only in subtle ways from object-based types such as `Integer` and `Boolean`. Primitive types can be automatically coerced (converted) to non-primitive versions or back again by the Gosu language in almost all cases. For example, from `int` to `Integer` or from `Boolean` to `boolean`.

You typically do **not** need to know the differences, and internally they are stored in the same type of memory location so there is no performance benefit to using primitives. Internally, primitives are stored as objects, and there is no speed improvement for using Gosu language primitives instead of their boxed versions, such as `int` compared to `Integer`.

The `boolean` type is a primitive, sometimes called an *unboxed type*. In contrast, `Boolean` is a class so `Boolean` is called a *boxed type* version of the `boolean` primitive. A boxed type is basically a primitive type wrapped in a shell of a class. These are useful so that code can make assumptions about all values having a common ancestor type `Object`, which is the root class of all class instances. For example, collection APIs require all objects to be of type `Object`. Thus, collections can contain `Integer` and `Boolean`, but not the primitives `int` or `boolean`.

However, there are differences while handling uninitialized values, because variables declared of a primitive type cannot hold the `null` value, but regular `Object` variable values can contain `null`.

For example, paste the following code into the Gosu Tester:

```
var unboxed : boolean = null  // boolean is a primitive type
var boxed : Boolean = null    // Boolean is an Object type, a non-primitive

print(unboxed)
print(boxed)
```

This code prints:

```
false
null
```

These differences are also notable if you pass primitives to `isAssignableFrom`. This method only looks at the type hierarchy and returns `false` if comparing primitives.

For example, paste the following code into the Gosu Tester:

```
var unboxed : boolean = true  // boolean is a primitive type
var boxed : Boolean = true    // Boolean is an Object type, a non-primitive

print((typeof boxed).IsPrimitive)
print((typeof unboxed).IsPrimitive)
print((typeof unboxed).isAssignableFrom( (typeof boxed) ))
```

This code prints:

```
false
true
false
```

In Gosu, the boxed versions of primitives use the Java versions. Because of this, in Gosu you find them defined in the `java.lang` package. For example, `java.lang.Integer`.

For more information about `Boolean` and `boolean`, see “Boolean” on page 66.

Java Type Reflection

Gosu implements a dynamic type system that is designed to be extended beyond its native objects. Do not confuse this with being *dynamically typed* because Gosu is *statically typed*. Gosu’s dynamic type system enables Gosu to work with a variety of different types.

These types include Gosu classes, Java classes, XML types, SOAP types, and other types. These different types plug into Gosu’s type system.

In almost all ways, Gosu does not care about the difference between a Java class or a native Gosu object. They are all exposed to the language through the same abstract type system so you can use Java types directly in your code. You can even extend Java classes, meaning that you can write Gosu types that are subtypes of Java types. Similarly, you can implement or even extend Java interfaces from Gosu.

The Gosu language transparently exposes and uses Java classes as Gosu objects through the use of Java `BeanInfo` objects. Java `BeanInfo` objects are analogous to Gosu’s `TypeInfo` information. They both encapsulate type metadata, including properties and methods on that type. All Java classes have `BeanInfo` information either explicitly provided with the Java class or can be dynamically constructed at runtime. Gosu examines a Java class’s `BeanInfo` and determines how to expose this type to Gosu. Because of this, your Gosu code can use the Gosu reflection APIs discussed earlier in this section with Java types.

Note: For a related topic, see “Java and Gosu” on page 235.

Type System Class

You can use the class `gw.lang.reflect.TypeSystem` for additional supported APIs for advanced type system introspection. For example, its `getByFullName` method can return a `Type` object from a `String` containing its fully-qualified name.

For example, the following code gets a type by a `String` version of its fully-qualified name and instantiates it using the type information for the type:

```
var myFullClassName = "com.mycompany.MyType"
var type = TypeSystem.getByFullName( myFullClassName )
var instance = type.TypeInfo.getConstructor( null ).Constructor.newInstance( null )
```

Compound Types

To implement some other features, Gosu supports a special kind of type called a *compound type*. A compound type combines one base class and additional interfaces that the type supports. You can declare a variable to have a compound type. However, typical usage is only when Gosu automatically creates a variable with a compound type.

For example, suppose you use the following code to initialize list values:

```
var x : List<String> = {"a", "b", "c"}
```

Note: The angle bracket notation indicates support for parameterized types, using Gosu generics features. For more information, refer to "Gosu Generics" on page 173.

You could also use this syntax using the `new` operator:

```
var x = new List<String>(){"a", "b", "c"}
```

Gosu also supports an extremely compact notation that does not explicitly include the type of the variable:

```
var x = {"a", "b", "c"}
```

It might surprise you that this last example is valid Gosu and is typesafe. Gosu infers the type of the `List` to be the least upper bound of the components of the list. In the simple case above, the type of the variable `x` at compile time is `List<String>`. If you pass different types of objects, Gosu finds the most specific type that includes all of the items in the list.

If the types implement interfaces, Gosu attempts to preserve the commonality of interface support in the list type. This means your list acts as expected with APIs that rely on support for the interface. In some cases, the resulting type is a *compound type*, which combines the following into a single type:

- **at most** one *class*
- one or more *interfaces*

For example, the following code initializes an `int` and a `double`:

```
var s  = {0, 3.4}
```

The resulting type of `s` is `ArrayList<java.lang.Comparable & java.lang.Number>`. This means that it is an array list of the compound type of the class `Number` and the interface `Comparable`.

Note: The `Number` class does not implement the interface `Comparable`. If it did, then the type of `s` would simply be `ArrayList<java.lang.Number>`. However, since it does not implement that interface, but both `int` and `double` implement that interface, Gosu assigns the compound type that includes the interfaces that they have in common.

This new compound type with type inference works with maps, as shown in the following examples:

```
var numbers = {0 -> 1, 3 -> 3.4}
var strings = {"a" -> "value"}
```

This also works with sets, as shown in the following example:

```
var s : Set = {1,2,3}
```

Compound Types using Composition (Delegates)

Gosu also creates compound types in the special case of using the delegate keyword with multiple interfaces. For more information, see "Using Gosu Composition" on page 151.

Type Loaders

The Gosu type system has an open type system. An important part of this is that Gosu supports custom type loaders. A type loader dynamically injects types into the language and attaches potentially complex dynamic behaviors to working with the type. A custom type loader adds types to the type system and optionally runs custom code every time any code accesses properties or call methods on them.

There are several built-in type loaders:

- **Gosu XML/XSD type loader.** This type loader supports the native Gosu APIs for XML. For more information, see "Gosu and XML" on page 195.
- **Gosu SOAP/WSDL type loader.** This type loader supports the native Gosu APIs for the web services SOAP protocol. This works through a Gosu type loaders that reads web service WSDL files and lets you interact with the external service through a natural syntax and type-safe coding. For more information, see "Calling WS-I Web Services from Gosu" on page 225.
- **Property file type loader.** This type loader finds property files in the hierarchy of files on the disk along with your Gosu class files. Gosu creates types in the appropriate package (by the property file location) for each property. You can access the properties directly in Gosu in a type-safe manner. For more information, see "Properties Files" on page 277.

You do not need to do anything special to install or enable these type loaders. Gosu includes these type loaders automatically for all Gosu code.

A future Gosu release will include documentation and supported APIs for creating custom type loaders.

chapter 23

Running Local Shell Commands

You can run local command line programs from Gosu.

Running Command Line Tools from Gosu

You can run local command line programs from Gosu. These APIs execute the given command as if it had been executed from the command line of the host operating system.

The Gosu class `gw.util.Shell` provides methods to run local command-line programs. For example, it can run `cmd.exe` scripts on Windows or `/bin/sh` on Unix. Gosu returns all content that is sent to standard out as a Gosu String. If the command finishes with a non zero return value, Gosu throws a `CommandFailedException` exception.

Content sent to standard error is forwarded to standard error for this Java Virtual Machine (JVM). If you wish to capture `StdErr` as well, use the `buildProcess(String)` method to create a `ProcessStarter` and call the `ProcessStarter.withStdErrHandler(gw.util.ProcessStarter.OutputHandler)` method.

IMPORTANT This method blocks on the execution of the command.

Pass the command as a `String` to the `exec` method.

For example:

```
var currentDir = Shell.exec( "dir" ) // windows
var currentDir = Shell.exec( "ls" )  // *nix
Shell.exec( "rm -rf " + directoryToDelete ) // directory remove on Unix
```

On windows, Gosu uses `CMD.EXE` to interpret commands. Beware of problems due to limitations of `CMD.EXE`, such as a command string may be too long for it. In these cases consider the `buildProcess(String)` method instead.

For related tools, see "Helpful APIs for Command Line Gosu Programs" on page 64 in the *Gosu Reference Guide*.

chapter 24

Checksums

This topic describes APIs for generating *checksums*. Longer checksums such as 64-bit checks sums are also known as *fingerprints*. Send these fingerprints along with data to improve detection from accidental modification of data in transit. For example, detecting corrupted stored data or errors in a communication channel.

This topic includes:

Overview of Checksums

To improve detection of accidental modification of data in transit, you can use *checksums*. A checksum is a computed value generated from an arbitrary block of digital source data. To check the integrity of the data at a later time, recompute the checksum and compare it with the stored checksum. If the checksums do not match, the data was almost certainly altered (either intentionally or unintentionally). For example, this technique can help detection of physical data corruption or errors in a communication channel.

Be aware that checksums cannot perfectly protect against intentional corruption by a malicious agent. A malicious attacker could modify the data so as to preserve its checksum value, or depending on the transport could substitute a new checksum for the modified data. To guard against malicious changes, use encryption at the data level (a cryptographic hash) or the transport level (such as SSL/HTTPS).

WARNING Checksums improve detection from accidental modification of data but cannot detect intentional corruption by a malicious agent. If you need that level of protection, use encryption instead of checksums, or in addition to checksums.

You can also use fingerprints to design caching and syncing algorithms that check whether data changed since the last cached copy. You can save the fingerprint of the cached copy and an external system can generate a fingerprint of its most current data. If you have both fingerprints, compare them to determine if you must resync the data. To work effectively, the fingerprint algorithm must provide near-certainty that a real-world change would change the fingerprint. In essence, a fingerprint uniquely identifies the data for most practical purposes, although in fact collisions (changed data with matching fingerprints) is theoretically possible.

Gosu provides support for 64-bit checksums in the class `FP64` in the package `gw.util.fingerprint`.

The `FP64` class provides methods for computing 64-bit fingerprints of the following kinds of data:

- `String` objects
- character arrays
- byte arrays
- input streams

This implementation is based on an original idea of Michael O. Rabin, with refinements by Andrei Broder. Fingerprints provide a probabilistic guarantee that defines a mathematical upper bound on the probability of a collision (a collision occurs if two different strings have the same fingerprint). Using 64-bit fingerprints, the odds of a collision are extremely small. The odds of a collision between two randomly chosen texts a million characters long are less than 1 in a trillion.

Suppose you have a set `S` of `n` distinct strings each of which is at most `m` characters long. The odds of any two different strings in `S` having the same fingerprint is described by the following equation (`k` is the number of bits in the fingerprint):

```
(nm^2) / 2^k
```

For practical purposes, you can treat fingerprints as uniquely identifying the bytes that produced them. In mathematical notation given two `String` variables `s1` and `s2`,using the → symbol to mean "*implies*":

```
new FP64(s1).equals(new FP64(s2)) → s1.equals(s2)
```

Do not fingerprint the value of (the raw bytes of) a fingerprint. In other words, do not fingerprint the output of the `FP64` methods `toBytes` and `toHexString`. If you do so, due to the shorter length of the fingerprint itself, the probabilistic guarantee is invalid and may lead to unexpected collisions.

Creating Fingerprints

To create a fingerprint object, use the constructor to the FP64 object and pass it one of the supported objects:

An example of passing a `String` object:

```
var s = "hello"
var f = new FP64(s)
```

An example of passing a character array:

```
var s = "hello"
var ca : char[] = {s[0], s[1], s[2], s[3], s[4]}
var f = new FP64(ca)
```

Note: There is an alternate method signature that takes extra parameters for start position and length of the desired series of characters in the array.

An example of passing a byte array:

```
var ba = "hello".Bytes // or use "hello".getBytes(
var f = new FP64(ba)
```

Note: There is an alternate method signature that takes extra parameters for start position and length of the desired series of byes in the array.

An example of passing a stream:

```
var s = "testInputStreamConstructor"
new FP64(new ByteArrayInputStream(gw.util.StreamUtil.toBytes(s))));
```

An example of passing an input stream:

```
var s = "testInputStreamConstructor"
new FP64(new StringBuffer(g));
```

An example of passing another `FP64` fingerprint object to the constructor to duplicate the fingerprint:

```
var s = "hello"
var f = new FP64(s)
var f2 = new FP64(f)
```

How to Output Data Inside a Fingerprint

To generate output data from a finger print, use the `FP64` method `toBytes()`, which returns the value of this fingerprint as a newly-allocated array of 8 bytes.

Instead of the no-argument method, you can also use the alternate method signature that takes a byte array buffer and the method writes the bytes there. The buffer must have length at least 8 bytes.

Alternatively, you can use a method `toHexString()`. This method returns the fingerprint as an unsigned integer encoded in base 16 (hexadecimal) and padded with leading zeros to a total length of 16 characters.

Extending Fingerprints

This class also provides methods for *extending* an existing fingerprint by more bytes or characters. This is useful if you are sure the only change to the source data was appending a known series of bytes to the **end** of the original `String` data.

Extending the fingerprint of one `String` by another `String` produces a fingerprint equivalent to the fingerprint of the concatenation of the two String objects. Given the two `String` variables `s1` and `s2`, this means the following is true:

```
new FP64(s1 + s2).equals( new FP64(s1).extend(s2) )
```

All operations for extending a fingerprint are **destructive**. In other words, they modify the fingerprint object directly (*in-place*). All operations return the resulting `FP64` object, so you can chain method calls together, such as the following:

```
new FP64("x").extend(foo).extend(92))
```

If you want to make a copy of a fingerprint, use the FP64 constructor and pass the FP64 object to copy:

```
var original = new FP64("Hello world")
var copy = new FP64(original) // a duplicate of the original fingerprint
```

chapter 25

Concurrency

This topic describes Gosu APIs that protect shared data from access from multiple threads.

This topic includes:

Overview of Thread Safety and Concurrency

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

The most common situation that requires proper concurrency handling is data in class static variables. Static variables are variables that are stored once per class rather than once per instance of the class. If multiple threads on the same Java virtual machine access this class, you must ensure that any simultaneous access to this data safely gets or sets this data.

If you are experienced with multi-threaded programming and you are certain that static variables or other shared data is necessary, you must ensure that you *synchronize* access to static variables. Synchronization refers to locking access between threads to shared resources such as static variables.

There are other special cases in which you must be particularly careful. For example, if you want to manage a single local memory cache that applies to multiple threads, you must carefully synchronize all reads and writes to shared data.

WARNING Static variables can be extremely dangerous in a multi-threaded environment. Using static variables in a multi-threaded environment can cause problems in a production deployment if you do not properly synchronize access. If such problems occur, they are extremely difficult to diagnose and debug. Timing in an multi-user multi-threaded environment is difficult, if not impossible, to control in a testing environment.

Gosu provides the following types of concurrency APIs to make it easy for you to write thread-safe code:

- **Scoping classes (pre-scoped maps).** Scope-related utilities in the class `gw.api.web.Scopes` help synchronize and protect access to shared data. These APIs return `Map` objects into which you can get and put data using different scope semantics. Gosu automatically synchronizes the `Map` objects to provide proper concurrent access semantics. For more information, see "Gosu Scoping Classes (Pre-scoped Maps)" on page 270
- **Lazy concurrent variables.** The `LazyVar` class (in `gw.util.concurrent)` implements what some people call a *lazy variable*. This means Gosu constructs it only the first time some code uses it. Because the `LazyVar` class uses the Java concurrency libraries, access to the lazy variable is thread-safe. The `LazyVar` class wraps the double-checked locking pattern in a typesafe holder. For more information, see "Concurrent Lazy Variables" on page 271
- **Concurrent cache.** The `Cache` class (in `gw.util.concurrent)` declares a cache of values you can look up quickly and in a thread-safe way. It declares a concurrent cache similar to a Least Recently Used (LRU) cache. Because the `Cache` class uses the Java concurrency libraries, access to the concurrent cache is thread-safe. For more information, see "Concurrent Cache" on page 272.

  WARNING Caches are difficult to implement and use. Caches can cause subtle problems. Use caches only as a last result for performance. If you use a cache, it is best to request multiple people on your team carefully review cache-related code.

- **Support for Java monitor locks, reentrant locks, and custom reentrant objects.** Gosu provides access to Java-based classes for monitor locks and reentrant locks in the Java package `java.util.concurrent`. Gosu makes it easier to access these classes with easy-to-read `using` clauses that also properly handle cleanup if exceptions occur. Additionally, Gosu makes it easy to create custom Gosu objects that support an easy-to-read syntax for reentrant object handling. For more information, see "Concurrency with Monitor Locks and Reentrant Objects" on page 273.

Gosu Scoping Classes (Pre-scoped Maps)

Gosu provides scope-related utility methods in the class `gw.api.web.Scopes`. These static methods help synchronize and protect access to shared data using synchronized `Map` objects that follow standard web-application scoping semantics.

IMPORTANT These methods are available only in execution contexts that are associated with a web request. If you attempt to accessed these methods in other contexts, Gosu throws an `IllegalStateException` exception. Be aware that how this data is stored is dependent on the application server container in which your application runs. Your data must satisfy any constraints that container imposes. For example, some application containers might require that your objects are serializable (implement the `Serializable` interface).

Call methods that correspond to different scopes:

Scope	Meaning	Method	Description
Request scope	A single thread-local request.	`getRequest`	Returns a `Map` to store and retrieve values whose lifespan is the lifespan of the request. This map is not synchronized since multiple threads typically cannot get to the same request object. You could create unexpected situations by passing this object to other threads, so you must avoid such actions.
Session	One web session	`getSession`	Returns a `Map` to store and retrieve values whose lifespan is the lifespan of the users session. This map is automatically synchronized since multiple threads can access the session simultaneously. For example, web AJAX requests.
Application	The entire Gosu application, including all requests and subthreads.	`getApplication`	Returns a `Map` to store and retrieve values whose lifespan is the lifespan of the web application. This is almost identical to static variables, but Gosu clears the map if a servlet shuts down and is later restarted.

For example, the following Gosu class creates an application-scoped variable.

```
class MyClass {

  // lazy variable using a block that calls a resource-intensive operation that retuns a String
  static var _data : java.util.Map

  construct() {

    // create an instance of a thread-safe shared Map with application scope
    _data = gw.api.web.Scopes.getApplication()

   // set variable in our scoped object. The object ensures any access is thread-safe.
    _data["Name"] = "John Smith"
  }
}
```

Concurrent Lazy Variables

In addition to using the Java native concurrency classes, Gosu includes utility classes that add additional concurrency functionality. The `LazyVar` class implements what some people call a *lazy variable*. This means Gosu constructs it only the first time some code uses it. Because the `LazyVar` class uses the Java concurrency libraries, access to the lazy variable is thread-safe. The `LazyVar` class wraps the double-checked locking pattern in a typesafe holder.

In Gosu, you will see the `make` method signature `LazyVar.make(gw.util.concurrent.LazyVar.LazyVarInit)` method signature, which returns the lazy variable object. This method requires a Gosu block that creates an object. Gosu runs this block on the first **access** of the `LazyVar` value. An example is easier to understand than the method signature. The following example passes a block as an argument to `LazyVar.make(...)`. That block creates a new `ArrayList` parameterized to the `String` class:

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

As you can see, the parameter is a block that creates a new object. In this case, it returns a new `ArrayList`. You can create any object. In real world cases this block might be very resource-intensive to create (or load) this object.

It is best to let Gosu infer the proper type of the block type or the result of the make method, as shown in this example. This simplifies your code so that you do not need to use explicit Gosu generics syntax to define the type of the block itself, such as the following verbose version:

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

To use the lazy variable, just call its `get` method:

```
var i = _lazy.get()
```

If the Gosu has not yet run the block, it does when you access it. If Gosu has run the block, it simply returns the cached value and does not rerun the block.

A good approach to using lazy variables is to define it as a static variable and then define a property accessor function to abstract away the implementation of the variable. The following is an example inside a Gosu class definition:

```
class MyClass {

  // lazy variable using a block that calls a resource-intensive operation that retuns a String
  var _lazy = LazyVar.make( \-> veryExpensiveMethodThatRetunsAString() )

  // define a property get function that gets this value
  property get MyLazyString() : String {
    return _lazy.get()
  }
}
```

If any code accesses the property `MyLazyString`, Gosu calls its property accessor function. The property accessor always calls the get method on the object. However, Gosu only runs the very expensive method once, the first time someone accesses the lazy variable value. If any code accesses this property again, the cached value is used. Gosu does not execute the block again. This is useful in cases where you want some system to come up quickly and only pay incremental costs for resource-intensive value calculations.

Concurrent Cache

A similar class to the `LazyVar` class is the `Cache` class. It declares a cache of values you can look up quickly and in a thread-safe way. It declares a concurrent cache similar to a Least Recently Used (LRU) cache. Because the `Cache` class uses the Java concurrency libraries, access to the cache is thread-safe.

To create a thread-safe cache

1. Decide the key and value types for your cache based on input data. For example, perhaps you want to pass a `String` and get an `Integer` back from the cache.
2. Use the key and value types to parameterize the Cache type using Gosu generics syntax. For example, if you want to pass a `String` and get an `Integer` back from the cache, create a new `Cache<String, Integer>`.
3. In the constructor, pass the following arguments:
   - a name for your cache as a `String` - the implementation uses this name to generate logging for cache misses
   - the size of your cache, as a number of slots
   - a block that defines a function that calculates a value from an input value. Presumably this is a resource-intensive calculation.

   For example,

   ```
   // A cache of string values to their upper case values
   var myCache = new Cache<String, String>( "My Uppercase Cache", 100, \ s -> s.toUpperCase() )
   ```
4. To use the cache, just call the get method and pass the input value (the key). If the value is in the cache, it simply returns it from the cache. If it is not cached, Gosu calls the block and calculates it from the input value (the key) and then caches the result. For example:

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

   This prints:

   ```
   "HELLO WORLD"
   "HELLO WORLD"
   ```

   In this example, the first time you call the `get` method, it calls the block to generate the upper case value. The second time you call the `get` method, the value is the same but Gosu uses the cached value. Any times you call the `get` method later, the value is the same but Gosu uses the cached value, assuming it still in the cache. If too many items were added to the cache and your desired item is unavailable, Gosu reruns the block to regenerate the value. Gosu then caches the result.

Alternatively, if you want to use a cache within some other class, you can define a static instance of the cache. The static variable definition itself defines your block. Again, because the `Cache` class uses the Java concurrency libraries, it is thread-safe. For example, in your class definition, define a static variable like this:

```
static var _upperCaseCache = new Cache<Foo, Bar>( 1000, \ foo -> getBar( foo ) )
```

To use your cache, your class can get a value from the cache using code like the following. In this example, `inputString` is a `String` variable that may or may not contain a `String` that you used before with this cache:

```
var fastValue = _upperCaseCache.get( inputString )
```

The first time you call the `get` method, it calls the block to generate the upper case value.

Any later times you call the `get` method, the value is the same but Gosu uses the cached value, assuming it still in the cache. If too many items were added to the cache and your desired item is unavailable, Gosu reruns the block to regenerate the value. Gosu then caches the result in the concurrent cache object.

An even better way to use the cache is to abstract the cache implementation into a property accessor function. Let the private static object `Cache` object (as shown in the previous example) handle the actual cache. For example, define a property accessor function such as:

```
static property get function UpperCaseQuickly( str : String ) {
    return _upperCaseCache.get( str )
  }
```

These are demonstrations only with a simple and non-resource-intensive operation in the block. Generally speaking, it is only worth the overhead of maintaining the cache if your calculation is resource-intensive combined with potentially repeated access with the same input values.

WARNING Caching can be difficult and error prone in complex applications. It can lead to run time errors and data corruption if you do not do it carefully. Only use caches as a last resort for performance issues. Because of the complexity of cache code, always have multiple experienced programmers review any cache-related code.

Concurrency with Monitor Locks and Reentrant Objects

From Gosu, you can use the Java 1.5 concurrency classes in the package `java.util.concurrent` to synchronize the variable's data to prevent simultaneous access to the data.

The simplest form is to define a static variable for a lock in your class definition. Next, define a property get accessor function that uses the lock and calls another method that performs the task you must synchronize. This approach uses a Gosu `using` clause with reentrant objects to simplify concurrent access to shared data.

For example:

```
...

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

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

...
```

The `using` statement automatically cleans up the lock, even if there code throws exceptions.

In contrast, this is a traditionally-structured verbose use of a lock using `try` and `finally` statements:

```
uses java.util.concurrent

...

static var _lock = new ReentrantLock()
  static var _someVar = ...

  property get SomeProp() : Object {
    _lock.lock()
    try {
      return _someVar.someMethod()
    } finally {
      _lock.unlock()
    }
  }
```

Alternatively, you can do your changes within Gosu blocks:

```
uses java.util.concurrent

...

property get SomeProp() : Object {
  var retValue : Object
  _lock.with( \-> {
    retValue = _someVar.someMethod()
  })
  return retValue
}
```

> **Note:** Although this approach is possible, returning the value from a block imposes some more restrictions on how you implement your `return` statements. Instead, it is usually better to use the `using` statement structure at the beginning of this topic.

The using statement version works with these lock objects because Gosu considers this objects *reentrant*.

Re-entrant objects are objects that help manage safe access to data that is shared by re-entrant or concurrent code execution. For example, if you must store data that is shared by multiple threads, ensure that you protect against concurrent access from multiple threads to prevent data corruption. The most prominent type of shared data is class *static variables*, which are variables that are stored on the Gosu class itself.

For Gosu to recognize a valid reentrant object, the object must have one of the following attributes:

- Implements the `java.util.concurrent.locks.Lock` interface. This includes the Java classes in that pacakge: `ReentrantLock`, `ReadWriteLock`, `Condition`.
- Casted to the Gosu interface `IMonitorLock`. You can cast **any** arbitrary object to `IMonitorLock`. This is useful to cast Java monitor locks to this Gosu interface. For more information about monitor locks, refer to:
  `http://en.wikipedia.org/wiki/Monitor_(synchronization)`
- Implements the Gosu class `gw.lang.IReentrant`. This interface contains two methods with no arguments: `enter` and `exit`. Your code must properly lock or synchronize data access as appropriate during the `enter` method and release any locks in the `exit` method.

For blocks of code using locks (code that implements `java.util.concurrent.locks.Lock`), a `using` clause simplifies your code.

The following code uses the `java.util.concurrent.locks.ReentrantLock` class using a longer (non-recommended) form:

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


function useReentrantLockOld() {
  _lock.lock()
  try {
      // do your main work here
    }
  finally {
    _lock.unlock()
  }
}
```

In contrast, you can write more readable Gosu code using the `using` keyword:

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


function useReentrantLockNew() {
  using( _lock ) {
    // do your main work here
  }
}
```

Similarly, you can cast any object to a monitor lock by adding "`as IMonitorLock`" after the object. For example, the following method call code uses itself (using the special keyword `this`) as the monitor lock:

```
function monitorLock() {
  using( this as IMonitorLock ) {
      // do stuff
    }
  }
```

This approach effectively is equivalent to a `synchronized` block in the Java language.

Assigning Variables Inside 'using' Expression Declaration

The using clause supports assigning a variable inside the declaration of the `using` clause.

This is useful if the expression that you pass to the `using` expression is both:

- something other than a single variable
- you want to reference it from inside the statement list inside the `using` clause declaration

For example, suppose you call a method that returns a file handle and you pass that to the `using` clause as the lock. From within the `using` clause contents, you probably want to access the file so you can iterate across its contents.

To simplify this kind of code, assign the variable before the expression using the `var` keyword:

```
using ( var VARIABLE_NAME = EXPRESSION ) {
  // code that references the VARIABLE_NAME variable
}
```

For example:

```
using( var out = new FileOutputStream( this, false ) ) {
  out.write( content )
}
```

Passing Multiple Items to the 'using' Statement

You can pass multiple items in the using clause expression. Separate each item by a comma character.

For example,

```
function useReentrantLockNew() {
  using( _lock1, _lock2, _lock3) {
    // do your main work here
  }
}
```

You can combine the multiple item feature with the ability to assign variables. For more about assigning variables, see "Assigning Variables Inside 'using' Expression Declaration" on page 275 .

For example:

```
using( var lfc = new FileInputStream(this).Channel,
       var rfc = new FileInputStream(that).Channel ) {

      var lbuff  = ByteBuffer.allocate(bufferSize)
      var rbuff  = ByteBuffer.allocate(bufferSize)

      while (lfc.position() < lfc.size()) {
        lfc.read(lbuff)
        rfc.read(rbuff)

        if (not Arrays.equals(lbuff.array(), rbuff.array()))
        {
          return true
        }
```

```
        lbuff.clear()
        rbuff.clear()
      }
      return false
    }
  }
```

Gosu ensures that all objects are properly cleaned up. In other words, for each object to create or resource to acquire, if it creates or acquires successfully, Gosu releases, closes, or disposes the object. Also note that if one of the resources fails to create, Gosu does not attempt to acquire other resources in later-appearing items in the command-seperated list. Instead, Gosu simply releases the ones that did succeed.

For more information about using clauses, see "Object Lifecycle Management ('using' Clauses)" on page 122 in the *Gosu Reference Guide*.

Note: For more information about concurrency and related APIs in Java, see:

`http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html`

chapter 26

Properties Files

Gosu includes automatic support for reading properties files in the Java properties format.

This topic includes:

Reading Properties Files

Gosu includes automatic support for reading properties files in the Java properties format. Gosu accomplishes this with a custom type loader that adds types in the type system for any file with the `.properties` file extension in the class hierarchy. The location of the file within the class hierarchy defines the package (namespace) for created types. Gosu creates a type that matches the name of the properties file without the file extension. The following procedure describes in detail how to use this feature.

To read a properties file from Gosu

1. Find your root of your class hierarchy.

   If your Gosu code is in a Gosu program (a `.gsp` file), you can add a root directory to your class path using the `classpath` statement. See "Setting the Class Path to Call Other Gosu or Java Classes" on page 60.

2. Decide where in your package hierarchy that you want to reference your properties file. For example, suppose the root of your class hierarchy is the path `/MyProject/gsrc`. If you want your properties file to be in the package `com.mycompany.config` and the properties file to be called `MyProps.properties`, create a new file at the path:

   ```
   /MyProject/gsrc/com/mycompany/config/MyProps.properties
   ```

3. In that file, add the following content:

   ```
   # the hash character as first char means the line is a comment
   ! the exclamation mark character as first char means the line is a comment

   website = http://gosu-lang.org/
   language = English

   # The backslash below tells the application to continue reading
   # the value onto the next line.
   ```

```
message = Welcome to \
          Gosu!

# Unicode support
tab : \u0009

# multiple levels of hierarchy for the key
gosu.example.properties.Key1 = Value1
```

A few things to notice:

- The `message` property definition uses multiple lines, using the backslash to continue reading from the next line.
- The tab property definition uses Unicode syntax with \u followed by four hexadecimal digits for the Unicode code point.
- The last property in the file uses multiple levels of hierarchy

4. To test this code from another Gosu class, use the following code:

```
uses com.mycompany.config.*

print("accessing properties...")
print("")

print(" message: ${MyProps.message}")
print(" website: ${MyProps.website}")
print(" gosu.example.properties.Key1: ${MyProps.gosu.example.properties.Key1}")
print(" unicode support (tab char): before${MyProps.tab}after")
```

Run this code to print the following:

```
accessing properties...

 message: Welcome to Gosu!
 website: http://gosu-lang.org/
 gosu.example.properties.Key1: Value1
 unicode support (tab char): before     after
```

To test this code with a Gosu program instead of a Gosu class, create a Gosu program called `PropsTest.gsp` one level higher than the root of your class hierarchy. Add a `classpath` statement to add the root of the class hierarchy to the class path. See "Setting the Class Path to Call Other Gosu or Java Classes" on page 60.

Limitations of the Properties File Type Loader

The properties file type loader does not support key values with spaces, or any other characters that would be illegal in a Gosu property name. Gosu omits any such properties.

For example, the following Java property file includes a key with a name that includes embedded spaces using the backslash character before each space character,

```
# Add spaces to the key
key\ with\ spaces = This is the value that could be looked up with the key "key with spaces".
```

Although it is a legal Java property, Gosu does not provide programmatic access to it.

chapter 27

Coding Style

This topic lists some recommended coding practices for the Gosu language. These guidelines encourage good programming practices that improve Gosu readability and encourage code that is error-free, easy to understand, and easy to maintain by other people.

This topic includes:

General Coding Guidelines

Omit Semicolons

Omit semicolons, as they are unnecessary in almost all cases. Gosu code looks cleaner this way.

Semicolons are only needed if separating multiple Gosu statements all written on one line within a one-line statement list. Even this is generally not recommended, although it is sometimes appropriate for simple statement lists declared in-line within Gosu block definitions.

Type Declarations

Omit the type declaration if you declare variables with an assignment. Instead, use "`as` *`TYPE`*" where appropriate. The type declaration is particularly redundant if a value needs coerce to a type already included at the end of the Gosu statement.

In other words, the recommended type declaration style is:

```
var delplans = currentPage as DelinquencyPlans
```

Do **not** add the redundant type declaration:

```
var delplans : DelinquencyPlans = currentPage as DelinquencyPlans
```

The == and != Operator Recommendations and Warnings

The Gosu `==` and `!=` operators are safe to use even if one side evaluates to `null`.

Use these operators where possible instead of using the `equals` method on objects. This protection with `null` is called *null-safety*.

Notice that Gosu's == operator is equivalent to the object method `equals` (`obj1.equals(obj2)`) other than its difference in null-safety.

> **Note:** For those who use the Java language also, the null-safety of the Gosu == operator is similar to the null-safety of the Java code `ObjectUtil.equals(...)`. In contrast, for both the Gosu and Java languages, the object method *`myobject`*`.equals(...)` is not null-safe.

So, any Gosu code that use the `equals` method, such as:

```
( planName.equals( row.Name.text ) )
```

...can be written in easier-to-understand code as:

```
( planName == row.Name.text) )
```

Although the == and the `!=` comparison operators are more powerful and more convenient than `equals()`, be aware of coercions that may occur. For example, because expressions adhere to Gosu's implicit coercion rules, the expression `1 == "1"` evaluates to `true`. In other words, the number 1 and the string representing the number 1 is true. This is because of implicit coercion that allows the string `"1"` to be assigned to an integer variable as the integer 1 without explicit casting.

While coercion behavior is convenient and powerful, it can be dangerous if used carelessly. Gosu produces compile warnings for implicit coercions. Take the warnings seriously and in most cases *explicitly* cast using the `as` keyword in cases that you want the coercion. Otherwise, fix the problem by rewriting in some other way entirely.

For example, an expression equates a date value with a string representation of a date value:

```
(dateVal == strVal)
```

It is safest to rewrite this as the following:

```
(dateVal == strVal as DateTime)
```

Carefully consider any implicit direct coercions that might occur with the == operator, and explicitly define coercions where possible.

If comparing array equality with the == and `!=` operators, Gosu does not let you compare *incompatible* array types. For example, the following code generates a compile time error because arrays of numbers and strings are incompatible:

```
new Number[] {1,2} == new String[] {"1","2"}
```

However, if the array types are comparable, Gosu recursively applies implicit coercion rules on the array's **elements**. For example, the following code evaluates to `true` because a `Number` is a subclass of `Object`, so the Gosu compares the individual elements of the table:

```
new Number[] {1,2} == new Object[] {"1","2"}
```

> **WARNING** Be careful if comparing arrays. Note the recursive comparison of individual elements for compatible array types.

For more information about the difference between == and === operators in Gosu, see "=== Operator Compares Object Equality" on page 83

Gosu Case Sensitivity Implications

Gosu compiles and run faster if you write all your Gosu as case-sensitive code. It is the standard Gosu style to always use the proper capitalization precisely as defined for all of the following:

- proper type names
- variable names
- keywords (such as `var` and `if`)

- method names
- property names
- package names
- all other language elements.

In addition to the performance inmprovement, using proper capitalization makes your code easier to read.

For more examples and additional information on this topic, see “Gosu Case Sensitivity” on page 33.

Class Variable and Class Property Recommendations

Always prefix `private` and `protected` class variables with an underscore character (_).

Avoid *public variables*. Convert public variables to properties, so that the interface to other code (the property names) is separated from the implementation (the storage and retrieval).

Although Gosu supports public variables for compatibility with other languages, the standard Gosu style is to use public properties backed by private variables rather than public variables. You can do this easily in Gosu on the same line as the variable definition using the **as** keyword followed by the **property name**.

In other words, in your new Gosu classes that define class variables, use this variable declaration syntax:

```
private var _firstName : String as FirstName
```

This declares a private variable called `_firstname`, which Gosu exposes as a public property called `FirstName`.

Do not do this:

```
public var FirstName : String     // do not do this. Public variable scope is not Gosu standard style
```

For more information about defining properties, see “Properties” on page 130.

IMPORTANT For Gosu classes data fields, the standard Gosu style is to use public properties backed by private variables rather than public variables. Do not use public variables in new Gosu classes. See “Properties” on page 130 for more information.

Use ‘typeis’ Inference

To improve the readability of your Gosu code, Gosu automatically downcasts after a `typeis` expression if the type is a subtype of the original type. This is particularly valuable for `if` statements and similar Gosu structures. Within the Gosu code bounded by the `if` statement, you do not need to do casting (“`as` *`TYPE`*” expressions) to that subtype. Because Gosu confirms that the object has the more specific subtype, Gosu implicitly considers that variable’s type to be the **subtype**, at least within that block of code.

The structure of this type looks like the following:

```
var VARIABLE_NAME : TYPE_NAME

if (VARIABLE_NAME typeis SUBTYPE_NAME) {

  // use the VARIABLE_NAME as SUBTYPE_NAME without casting
  // This assumes SUBTYPE_NAME is a subtype of TYPE_NAME
}
```

For example, the following example shows a variable declared as an `Object`, but downcasted to `String` within the `if` statement in a block of code within an `if` statement.

Because of downcasting, the following code is valid:

```
var x : Object = "nice"
var strlen = 0

if( x typeis String ) {
  strlen = x.length
}
```

This works because the `typeis` inference is effective immediately and propagates to adjacent expressions.

It is important to note that `length` is a property on `String`, not `Object`. The downcasting from `Object` to `String` means that you do not need an additional casting around the variable `x`. In other words, the following code is equivalent but has an **unnecessary** cast:

```
var x : Object = "nice"
var strlen = 0

if( x typeis String ) {
  strlen = (x as String).length // "length" is a property on String, not Object
}
```

Use automatic downcasting to write easy-to-read and concise Gosu code. Do not write Gosu code with unnecessary casts. For more information, see “Automatic Downcasting for ‘typeis’ and ‘typeof’” on page 254 in the *Gosu Reference Guide*.