GWiz Programming Language Reference Manual

Katherine Duff kpd2128, Ashley Kim atk2141,
Elisa Luo eyl2130, Rebecca Yao rby2107

February 24, 2021

Contents

1 Overview of GWiz

GWiz is a class based, imperative, object oriented programming language that allows for the analysis of GPX files. GPX files are generated by using a watch, phone, or other device to track a run, walk, swim, hike, or bike ride.

Inspired by Java, GWiz is strongly and statically typed to help differentiate between compile time and run time errors as well as to optimize language performance. The GWiz programming language dynamically allocates and leaks memory, unlike languages like C which require explicit allocation and deallocation of memory.

Most notable to GWiz is the Activity class built into Gwiz's standard library. Each GPX file can be represented as an Activity type through the structure of a linked list.

2 Lexical Conventions

2.1 Tokens

After lexical translation, characters are reduced to a sequence of input elements, those being white space, comments, and tokens. Tokens are divided into identifiers, separators, operators, GWiz keywords, and literals.

2.2 Comments

GWiz has both single-line and multi-line comments. They are both denoted using the /* and */ symbols. Comments start with the /* characters and end with the */ characters, ignoring all characters between the start and ending characters.

```
/* This is a single line comment. Text in here is ignored */
```

```
/* This is a multi line comment.
Text in here
is ignored */
```

Comments cannot be nested.

2.3 Identifiers

An identifier in GWiz is an unlimited length sequence of ASCII letters [A-Z, a-z] and decimal digits [0-9]. It must begin with a letter, cannot be a GWiz keyword, or contain a character other than a letter or digit (e.g. student_name is not a valid identifier)

Convention for identifiers is camel case, where the first letter is lowercase and every subsequent first character of a word is uppercase. (e.g. studentName)

The regex for identifiers following GWiz convention is `[a-z][0-9a-zA-Z]*`

2.4 Separators

The following 9 tokens are the separators in GWiz.

```
( ) { } [ ] ; , .
```

- () is used for precedence in expressions and denoting arguments in method calls

- { } is used for method and class scoping

  is used for array indices

- ; denotes the end of a line and separates statements
- , is used to separate variables or parameters
- . is used to separate a variable or method from a reference variable

2.5 Operators

The following 14 tokens are the operators in GWiz. These are the arithmetic, assignment, comparison, and logical operators in GWiz.

```
+ - * /
=
== < > != >= <=
! && ||
```

2.6 Keywords

The following identifiers are reserved for use as keywords and may not be used as identifiers.

```
if, else, for, return
void, class, this
int, double, char, boolean, string
```

Note, "true/"false", and "null" are not keywords in GWiz, but are boolean literals and the null literal, respectively.

2.7 Literals

A literal is a representation of a value of a primitive type, the String type, or the null type.

2.7.1 Integer Literals

A sequence of one or more numerical digits representing an integer. The matching regex is `[-]?[0-9]+`
Examples of integer literals:

```
0
5
163
-42
```

2.7.2 Double Literals

A sequence of zero or more numerical digits followed by a '.', followed by one or more numerical digits. The matching regex is `[-]?[0-9]*'.'[0-9]+`
Examples of double literals:

```
1.4
.5
0.5829
-1.5
```

2.7.3 Character Literals

A single character enclosed by a pair of single quotation marks. The matching regex is `[a-zA-Z]`
Examples of character literals:

```
char ch = 'c';
char chCapital = 'C';
```

2.7.4 Boolean Literals

Boolean types represent true and false. They are represented as `true` and `false` in GWiz.
Boolean literals:

```
true
false
```

2.7.5 String Literals

A sequence of characters enclosed in double quotes. Escape sequences may be present between the double quotes.
Examples of string literals:

```
String str = "hello world!";
String c = "c";
String empty = "";
```

2.8 The Null Literal

The null type has one value, the null reference, represented by `null`.

3 Data Types

There are two kinds of data values that can be stored in variables, passed as arguments, returned by methods, and operated on: primitive values and reference values.

3.1 Primitives

The primitive types are the boolean type and the numeric type. The numeric types are the integral types int and char, and the floating point type double.

- int: 4 bytes, 2's complement. From -2147483648 to 2147483647, inclusive
- char: 2 bytes, stores a single ASCII character. From '\u0000' to '\uffff' inclusive, that is, from 0 to 65535
- double: 8 bytes, IEEE 754 floating point value.
- boolean: 1 byte, 00000001 for true, 00000000 for false. Booleans will default to false unless otherwise assigned.

3.2 Reference Types

The reference types are class types, type variables, and array types.

3.3 Null Type

The null type is a special type, of the expression `null`. Using the null reference is the only possible value of an expression of the null type.

3.4 Mutability

In GWiz, all primitive objects are mutable, including ints, doubles, arrays of chars, and booleans. Strings are not primitives and are the only immutable object in GWiz. User defined types and standard library types are mutable, so modifying them does not overwrite the underlying object.

3.5 User Defined Types

Users can define their own types through classes. Classes are declared using the standard Java syntax. There is no inheritance.

```
1 class Person {
2 /* instance variables */
3 String name;
4 int age;
5
6 /*constructor*/
7 Person(String name, int age){
```

```
this.name = name;
this.age = age;
}

/*methods*/
String getName(){
   return name;
}
}
```

3.6 Standard Library Types

GWiz provides a number of built-in types. This will be expanded upon in a later section, but are implemented in GWiz as objects and have associated methods and attributes. Strings in GWiz are immutable and have a plethora of operations.

3.7 Memory

GWiz dynamically allocates and leaks memory. It does not require explicit memory allocation or management. Objects are passed by value, not by reference.

4 Type System

4.1 Explicit Typing

Variable declarations, parameters, and return values must be associated with an explicit type. Variable type is denoted by a type specifier which precedes a variable name. Type specifiers include int, char, string, float/double, or a user-defined type.

5 Expressions and Operators

An expression consists of at least one operand and zero or more operators. Available operators in GWiz are detailed in the following subsections.

5.1 Unary Operators

Unary operators act on an expression. In GWiz, NEG and NOT are the two unary operators. NEG is denoted by the symbol - and indicates the negation of an integer or float. NOT is denoted by the symbol ! and indicates the negation of a boolean expression.

5.2 Binary Operators

Binary operators act on two expressions. Examples of binary operators in GWiz include: +, -, *, /, =, ==,, !, ¡, ¿, ¡=, ¿=, &&, ——

1. Arithmetic Operators

   (a) Addition is performed on two values of the same type. Two strings can also be concatenated using the addition operator.

```
2 + 3 /* Evaluates to 5 */
1.2 + 5.0 /* Evaluates to 6.2 */
"hello" + "world" /* Evaluates to "helloworld" */
```

   (b) Subtraction is performed on two values of the same type.

```
10 - 3 /* Evaluates to 7 */
9.2 - 5.1 /* Evaluates to 4.1 */
```

   (c) Multiplication is performed on two values of the same type.

```
8 * 6 /* Evaluates to 48 */
2.2 * 1.1 /* Evaluates to 2.42 */
```

   (d) Division is performed on two values of the same type.

```
10 / 2 /* Evaluates to 5 */
3.0 / 0.5 /* Evaluates to 6.0 */
```

2. Assignment Operator
   The assignment operator, denoted by =, stores a value in a variable. The variable appears on the left of the = and the value to store appears on the right.

```
x = 3.0; /* The value 3.0 is stored in the variable x */
```

3. Relational Operators
   Relational operators determine how two operands relate to each other. In GWiz, relational operators include equal to, not equal to, greater than, and less than, denoted by ==, !=, ¿, ¡, ¿=, ¡= respectively. An expression containing two inputs and a relational operator returns true or false.

```
x = 0;
y = 1;
x > y /* Evaluates to false */
x == y /* Evaluates to false */
x != y /* Evaluates to true */
```

5.3 Operator Precedence

Operator precedence from lowest to highest precedence:

Operator	Meaning	Associativity
;	Statement end	Left
=	Assignment	Right
.	Access	Left
||	OR	Left
&&	AND	Left
= !=	Equality/Inequality	Left
><>=<=	Comparison	Left
+ -	Addition/Subtraction	Left
*/	Multiplication/Division	Left
type(variable)	Casting	Left
!	NOT	Right
-	NEGATION	Right

6 Statements

In GWiz, a statement is one of the following:

- expression
- return statement
- if statement
- if else statement
- for loop

6.1 If else Statement

An if statement evaluates a condition (an expression) in parentheses to be true, the program will evaluate the statements demarcated by immediate curly braces. Otherwise, if the condition evaluates to false, the program will check for an else statement and execute the block of statements contained in the curly braces following else.

```
if (condition) {
   /* some statements */
} else {
   /* some other statements */
}
```

6.2 For Statement

For statements are used to iterate through a range of values. The statement must include an expression that initializes a looping variable, an expression that constrains that variable to indicate when to stop looping, and an expression that increments or changes the looping variable. Statements inside the for loop as many times as the variable is incremented and/or execute with the value of the looping variable.

```
for (int i=0; i<10; i=i+1) {
   /* some statements */
}
```

6.3 Return Statement

The return statement indicates the end of a function's execution and returns control to the function that called it. The type of the return value must match the return type explicitly named in the function declaration.

```
int foo() {
    int x = 2;
   return x; /* x is of type int */
}
```

7 Functions

7.1 Function Declarations

A function statement takes certain inputs as parameters and returns one value. The body of a function statement is delimited by curly braces, and the return type of the function value must be explicitly stated. The type returned must match with the expected return type. An example of a function declaration follows:

```
int foo(int x, int y) {
   return x+y;
}
```

7.2 Function Calls

A function is called by its identifier. Its arguments must be contained in parentheses and separated by commas. An example of a function call follows:

```
foo(2,3);
```

7.3 Variable assignment from functions

A function may be called as the right-hand side of a variable assignment. The variable would be assigned to the return value of the function as follows:

```
int x = foo(2,3);
```

8 Classes

GWiz supports basic classes without any inheritance. Classes can have instance variables, and class methods. Class variables are not supported, so each instantiation of a class has unique variables. All variables and methods are public and can be accessed from other classes. Each class must have a constructor: a method declared using the name of the class which returns a new instance of the class. The constructor has user-defined parameters. All classes are mutable.

Syntax for declaring a class with one instance variable of type integer follows:

```
class Test {
    int t1;

    Test(int t2) {
        t1 = t2;
    }

    int add(int t3) {
        return t1 + t3;
    }
}
```

Classes are instantiated as follows:

```
Test test1 = Test(3);
```

Class methods are called as follows:

```
test1.add(2); //returns 5
```

9 Standard Library

All methods and attributes in the GWiz Standard Library are mutable and public.

9.1 Array

Arrays in GWiz operate similarly to arrays in Java. Users must declare the type of the list and its length at instantiation. All elements of the array must be of the same type, and the length of the array cannot change. A new instance of an array can be defined as follows.

```
int[] list = new int[5]; //array of type int with length 5
```

Method/operator	Behavior
list[n]	Returns nth element from the list
list.length	Returns the number of elements in the list

Array methods can be called exactly as specified above.

9.2 Strings

Strings are a class implemented to handle text-based data. Strings are immutable, so any operation or method performed on a string will return a reference to a new String.

A String object is instantiated as follows:

```
String str = "hello";
```

Method/operator	Behavior
equals(String other)	Returns true if each element of the String has equal ASCII values, false otherwise
str1 + str2	Returns the concatenation of two operand Strings in a new String reference

The String methods listed above can be called as follows:

```
String str = "gee whiz";
str.equals("gee whiz"); // returns true
String s = str + " kid"; // "gee whiz kid"
```

9.3 print()

A call to the method print() sends a String representation of the operand to output. Primitive types have a defined print function which will be applied to x. If x is a reference to an object, then the .toString() method will be called if it is defined. Otherwise, the memory address of the object will be sent to output.

9.4 Casting

The user can explicitly cast integers into doubles, doubles into integers, and primitive types into Strings. The user can cast both literals and expressions. Expressions in the parentheses will be evaluated first before being cast to String. Checking for casting will be done at compile time.

```
double(int x) -> double
int(double x) -> int
String(int x) -> String
String(double x) -> String
String(char x) -> String
```

The following classes in the Standard Library are created with the purpose of simplifying the analysis and application of .gpx data.

9.5 GPXFile

A GPXFile is an abstract representation of file and directory path names. The original .gpx file cannot be modified. An invalid file path results in a compile time error.

A GPXFile can be instantiated as follows:

```
GPXFile file = GPXFile(/home/documents/Running/jan3.gpx);
```

All methods and attributes use dot notation for calling, access mutation.

```
file.filepath //access the filepath attribute
```

Constructor Parameter and Type	Description
filepath ->(String)	A String representing the absolute file path to the .gpx file

9.6 GPXScanner

A GPXScanner is a simple text scanner that can parse a GPXFile using regular expressions. GPXScanner will link to an external C library to parse a .gpx file. The parsed data will be used to create an Activity object. This class has no parameters.

A GPXScanner can be instantiated as follows:

```
GPXScanner g = GPXScanner();
```

All methods and attributes use dot notation for calling, access mutation.

```
Activity a = g.readGPX(file); // parses file and returns Activity
```

Method/Attributes	Behavior
readGPX(GPXFile file)	Parses the .gpx file associated with the GPXFile object

9.7 Coordinate

A Coordinate is a representation of a single point on Earth.

A Coordinate can be instantiated as follows:

```
Coordinate c = Coordinate(0, 0);
```

All methods and attributes use dot notation for calling, access mutation.

```
double long = c.longitude; //returns the longitude of this Coordinate
```

Constructor Parameter and Type	Description
longitude ->(double)	Longitude
latitude ->(double)	Latitude

Methods/Attributes	Behavior
distance(Coordinate other)	Returns the euclidean distance between the two points

9.8 DateTime

Representation of a single naive date and time. A DateTime can be constructed with either a UTCDateTime string, or one or more of its components which DateTime will use to construct a UTCDateTime representation.

A DateTime can be instantiated as follows:

```
DateTime d = DateTime(2020, 02, 24, 10, 3, 7.58);
```

All methods and attributes use dot notation for calling, access mutation.

```
int k = d.year; //access year attribute
```

9.9 TrackPoint

Representation of a single GPS waypoint (Coordinate and DateTime).

A TrackPoint can be instantiated as follows:

```
TrackPoint t = TrackPoint(c, d);
```

Constructor Parameter and Type	Description
UTCDateTime ->(String)	A String representing a UTC Date and Time
year ->(int)	A year represented as an int
month ->(int)	A month represented as an int
day ->(int)	A day represented as an int
hour ->(int)	An hour represented as an int
min ->(int)	A minute represented as an int
second ->(double)	A second represented as an int

All methods and attributes use dot notation for calling, access mutation.

```
double dist = t.distDiff(t1);
```

Constructor Parameter and Type	Description
coor ->(Coordinate)	Coordinate
dt ->(DateTime)	DateTime

Methods/Attributes	Behavior
timeDiff(TrackPoint other)	Returns a double representing the time difference between the two DateTimes in seconds.
distDiff(TrackPoint other)	Returns a double representing the Euclidean distance between the two Coordinates
next	A reference to the following TrackPoint

9.10 Activity

A singly linked list of TrackPoints representing a cumulative exercise activity. This class typically will be instantiated through the use of the GPXScanner's .readGPX(GPXFile file) method. Otherwise, the user can construct an Activity as defined.

An Activity can be instantiated as follows:

```
Activity a = Activity(t); //New Activity with TrackPoint t as the head
```

All methods and attributes use dot notation for calling, access mutation.

```
double dist = a.getRouteTime();
```

Constructor Parameter and Type	Description
head ->(TrackPoint)	The first TrackPoint in the linked list

Methods/Attributes	Behavior
getRouteTime()	Returns a double representing the total time of the Activity in seconds
getRouteDistance()	Returns a double representing the total point-to-point distance of the Activity
getEuclidDistance()	Returns a double representing the Euclidean distance from the first to last point.
last	A reference to the last TrackPoint in the Activity list

9.11 Math

Math methods can be called as follows:

Methods/Attributes	Behavior
sqrt(double d)	Returns a double representing the square root of the parameter
exp(double base, int power)	Returns a double representing the first parameter raised to the second power. get
square(int d); square(double d);	Returns a double representing the square of the input parameter.

10 Sample Code

10.1 Example 1

```
boolean greaterThanTwo (int a) {
   return (a > 2);
}
```

10.2 Example 2

```
int gcd (int a, int b) {
   int remainder = 0;
   while (a % b > 0) {
      remainder = a % b;
      a = b;
      b = remainder;
   }
   return b;
}
```

10.3 Compare Distance

This program takes two .gpx files and prints out the longer of the two distances travelled.

```
void main() {
// Create two GPXFile objects with .gpx files taken from a run
GPXFile jan30 = GPXFile(/home/documents/user2_01_30_2021.gpx);
GPXFile jan31 = GPXFile(/home/documents/user2_01_31_2021.gpx);

// Create GPXScanner object to parse the two GPXFiles
GPXScanner reader = GPXScanner();

// Parse the GPXFile using the GPXScanners method read_gpx()
Activity satRun = reader.readGPX(jan30);
Activity sunRun = reader.readGPX(jan31);

// Use the Activity objects distance method to get distances for each
double satDist = satRun.getRouteDistance();
double sunDist = sunRun.getRouteDistance();

    // Compare total distances and print distance of the longer activity
    if (satDist > sunDist) {
       print(satDist);
    } else {
       print(sunDist);
    }
}
```

10.4 Change Starting Point

This program parses a .gpx file, then changes the starting latitude, longitude, date, time and prints out the new total distance.

```
void main() {
   // Create a GPXFile object
   GPXFile file1 = GPXFile(/home/documents/user3_01_21_2021.gpx);

   // Create a GPXScanner object to parse the GPXFile
   GPXScanner reader = GPXScanner();

   // Parse the GPXFile using GPXScanners method read_gpx();
   Activity goldenRun = reader.readGPX(file1);

   // Create new TrackPoint object with a coordinate and DateTime
   Coordinate c = Coordinate(0, 0);
   DateTime dt = DateTime("2021-01-21T23:30:42Z");
   TrackPoint tp = TrackPoint(c, dt);

```

```
16    // Modify the head of the Activity to be the above TrackPoint
17    goldenRun.head = tp;
18
19    print(goldenRun.getRouteDistance());
20  }
```