2018

Project Emerald: Designing a Language to be Fun

Addison Bostian
Ouachita Baptist University

SENIOR THESIS APPROVAL

This Honors thesis entitled

"Project Emerald: Designing a Language to be Fun"

written by

Addison Bostian

and submitted in partial fulfillment of
the requirements for completion of
the Carl Goodson Honors Program
meets the criteria for acceptance
and has been approved by the undersigned readers.

Jeff Matocha, thesis director

Steve Hennagin, second reader

Johnny Wink, third reader

Dr. Barbara Pemberton, Honors Program director

April 24, 2018

Project Emerald

Ouachita Baptist University

Addison Bostian

April 24, 2018

Contents

Introduction

I designed the language described here to be, first and foremost, fun. I wanted it to be a programmer's go-to language, the language that you pick up for personal projects or utilities. I felt the way to make this happen was to make it fun to write in. In order to accomplish this, the language derives from several existing languages, taking what I believed were the best parts of each of them. Combining principles from multiple languages sounds like a good idea, I quickly ran into problems that would make developing a compiler extremely difficult, if not impossible. Because of this, I had to make choices between certain features. Therefore, this language is not exactly how I originally envisioned it; It represents, rather, my compromise between functionality and the compiler.

Chapter 1

Origins

1.1 The Idea

The idea for this language started when I was working on a project in Java. I got to a portion of the project that lent itself to the use of lambdas. However, in Java, lambdas are not a particularly easy thing to do, and I started wishing that I had started the project in Ruby, where lambdas are extremely easy. I considered starting again in Ruby but then realized that, if I changed to Ruby, while I would make certain things easier, I would also make other things harder. This made me start thinking about a language that had all of the features I was looking for.

At the time, I was in a Programming Languages class where we were talking about the beginnings of compilers. I realized that writing a compiler was not as far out of reach as I had originally thought, and it was something that I could conceivably do myself. Because of that, I started wondering if I could create my own language.

1.2 Language Influences

During the design process, I drew heavily from languages such as Ruby, Java, and C# because most of my programming had been done in those languages. Other influences included Python, Javascript, and F#. When I started I had several goals. I wanted the language to be object oriented, which would allow data and actions to be easily grouped together. I wanted functional programming in the language to be easy, the ability to pass functions as parameters is a very powerful tool. I wanted collection manipulation to be easy, as this is one the most common things to do when programming. I also wanted the language to be a hybrid language, meaning it would compile down to an intermediate language, and that compiled program could be distributed to multiple types of systems without recompilation.

1.2.1 Object Oriented

While all the languages that influenced me were object-oriented in some way, my favorite was the Ruby implementation: Everything is an object, including numbers and letters. This extreme object-oriented stance made the most sense to me because you must only learn one set of rules. For example, in Java there is often confusion associated with new programmers learning the difference between passing base types and passing objects as parameters, where as in a language such as Ruby, where everything is an object, you only have to learn how objects are passed. To eliminate this problem, as well as to facilitate some other ideas I had such as operator overloading, I chose to make everything an object.

1.2.2 Functional Programming

Functional programming, at a basic level, is the ability to pass functions as parameters, and evaluate them inside the function. One of my favorite things about languages like C#, Javascript, and Ruby is that they make functional programming exceptionally easy. Functional programming is not something I necessarily do all the time, but when a problem comes along that lends itself to functional programming, having a language that makes it easy is worth it. I feel like Java handles functional programming exceptionally badly. In Java there is too much extraneous syntax when using lambdas, and this usually makes it quicker and easier to approach whatever problem you are trying to solve in a different manner. I chose to model C# in regards to lambdas and functional programming. C# has much cleaner syntax when using lambdas, and their use feels very natural when programming.

1.2.3 Collection Manipulation

Much of programming deals with manipulating collections of objects, so one goal of emerald is to make these manipulations easy. Ruby and C# both have excellent collection manipulation capabilities, and are in fact extremely similar in the way they implement it. I choose to use the best parts of both, with a slightly bigger influence from Ruby than C#. Ruby and C# both have built in methods for manipulating collections. Ruby is dynamically typed, and therefore declaring a lambda and passing it to a function is extremely easy, with barely any syntax required. C# required slightly more syntax, and uses the '=¿' operator in these functions. I took the minimal syntax from ruby and combined it with the '=¿' operator from C#.

1.2.4 Hybrid Language

One of the downsides to Ruby is that it is interpreted, meaning evaluated at runtime, which makes Ruby programs relatively slow. On the other hand C is a compiled language, meaning evaluated at compile time, not runtime, which makes C programs extremely fast to run after compilation. Unfortunately, compilation hampers cross-platform compatibility. Java and C# fit between these two, and as such are hybrid languages. Hybrid languages provide a balance between C and Ruby, a program written in a hybrid language will have execution time and portability between those of C and Ruby.

They are faster then Ruby but slower than C, but they make cross-platform compatibility much easier than C. Based on my goals for the language, I chose to make it a hybrid language. I also chose to compile to IL, the Microsoft Intermediate Language, which is what all of the .NET languages compile to. This means that a program written in my language would be able to run anywhere a .NET application could, which is almost everywhere.

Chapter 2

Language Documentation

This chapter documents the syntax and semantics of Emerald. It should describe everything you need to start using the language, then through more advanced topics like collections, control structures, objects, lambdas, and collection manipulation.

2.1 Variables

A variable in Emerald holds a reference to an object. Therefore all variables are reference variables, which has implications for copying values, comparing values, and passing values as parameters. Variables are initialized by declaring the type of the variable followed by the name, for example `Int x;` creates a variable named x of type Integer. Declaration and assignment may also be done in one line (`e.g. Int x = 42;`)

2.2 Data Types

Programmers typically use many data types to represent information. Everything is an object in Emerald, we do not have base types, and therefore will not use that terminology to refer to them. Emerald's common types include: `Int, Char, Double, Float, Bool, Long, and String.`

When objects are assigned a value without the `new` keyword, Emerald invokes the objects copy constructor. `String str = "hello";` is converted to `String str = new String("hello");` because `String(String string)` is the string constructor that matches the list of parameters. This feature will be discussed further in the Objects section.

2.2.1 Int

The type Int represents an Integer, a whole number. It is also the base class for most other number types such as `Float` and `Double`. An Integer is a 32-bit value and by default can store any whole number from -2147483648 to 2147483647.

Initialization

An Integer can be initialized in any of the following ways.

```
Int a = 42;
Int b = new Int(42);
Int c = 47-12*2;
```

Multiple Integers can be initialized at once in the following ways.

```
Int a = 41, b = 42, c = 43; // a #=> 41, b #=> 42, c #=> 43
Int d, e, f = 42; // d #=> 42, e #=> 42, f #=> 42
```

Binary Operators

There are several operators available to the Int class such as: `+`, `-`, `/`, `*`, `%`, `**`, `==`. These operators all function in the normal mathematical way, with the double equals representing equality checking. These are binary operators which work on two operands of the Int type or a subclass of Int.

The `+` operator arithmetically adds its two operands numeric value together and evaluates to the result. The `+` operator is a binary infix operator. For example, after the following code has executed, c contains 42.

```
Int a = 35;
Int b = 7;
Int c = a + b; // c #=> 42
```

The `-` operator arithmetically subtracts the numeric value of operand 2 from the numeric value of operand 1 and evaluates to the result. The `-`

operator is a binary infix operator. For example, after the following code has executed, c contains 42.

```
Int a = 56;
Int b = 14;
Int c = a - b; // c #=> 42
```

The * operator arithmetically multiplies its two operands' numeric values together and evaluates to the result. The * operator is a binary infix operator. For example, after the following code has executed, c contains 42.

```
Int a = 7;
Int b = 6;
Int c = a * b; // c #=> 42
```

The / operator arithmetically divides the numeric value of operand 1 by the numeric value of operand 2 and evaluates to the result. The / operator is a binary infix operator. For example, after the following code has executed, c contains 42.

```
Int a = 126;
Int b = 3;
Int c = a / b; c #=> 42
```

The return type of / is the type of its most precise operand. An Int divided by and **Int** is always an **Int**, however and **Int** divided by a **Double** evaluates to a **Double**. The second operand may not be zero. For example, after the following code has executed, c contains 1, and c contains 1.5.

```
Int a = 12, b = 8;
Double c = a/b; //c #=> 1

Double d = 8;
Double e = a/d; //e #=> 1.5
```

The % operator is the modulus or mod operator. It evaluates to the remainder of the numeric value of operand 1 divided by the numeric value of operand 2. The % operator is a binary infix operator. For example, after the following code has executed, c contains 2.

```
Int a = 14, b = 3;
Int c = a % b; // c #=> 2
```

The ** operator raises the numeric value of operand 1 to the power of the numeric value of operand 2 and returns the result. The ** operator is a binary infix operator. For example, after the following code has executed, c contains 32.

```
Int a = 2, b = 5;
Int c = a**b; // #=> 32
```

Unary Operators

There are several unary operators available to the Int class such as ++, --. These unary operators work on a single Int.

The -- operator decrements the numeric value of the operand. The -- operator takes one operand in either of the following ways: operand-- and --operand, which are post-decrement and pre-decrement, respectively.

```
Int a = 43;
a--; //a #=> 42
```

When post-decrementing, the value of the variable is used before it is decrementing, when pre-decrementing, the value of the variable is used after it is decrementing.

```
Int x,y = 42;
Int a = --x; // a #=> 41
Int b = x--; // b #=> 42
```

The ++ operator increments the numeric value of the operand. The ++ operator takes one argument in either of the following ways: operand++ and ++operand, which are post-increment and pre-increment, respectively.

```
Int a = 41;
a++; // a #=> 42
```

When post-incrementing the value of the variable is used before it is incremented, when pre-incrementing the value of the variable is used after it is incremented.

```
Int x,y = 42;
Int a = ++x; // a #=> 43
Int b = x++; // b #=> 42
```

Methods

These methods are available to the Int class, as well as all subclasses of Int due to inheritance. Methods generally start with a lowercase letter, continuing in camel case. They are called with the conventional dot (.) operator, with parentheses after the name. For example, the following code uses the .copy() method to put a copy of x into y.

```
Int x = 42;
Int y = x.copy(); // y #=> 42
```

public void copy() The .copy() method returns a new instance of the Int class that is the same as the calling object.

```
Int x = 42;
Int y = x.copy();
// at this point x and y have the same value
x++;
// at this point x = 43, y = 42
```

public Bool equals(Int i) The .equals(Int i) method returns a Bool (documented lated) indicating whether or not the calling object and the parameter have equal values. This method returns false if the parameter is null.

```
Int x, y = 42;
Bool a = x.equals(y); // a = true

Int z = 43;
Bool b = x.equals(z); // b = false

Int i;
Bool c = x.equals(i); //c = false
```

public String toString() The .toString() method returns a String (documented later) containing the numeric value of the Int.

Properties

Int's properties are read and write, and are accessed with the dot (.) operator, similar to methods, with the difference being that properties never have parentheses at the end or parameters. Properties generally start with a capital letter and continue in camel case.

public Bool PositiveOnly: PositiveOnly is used to set he range of values a particular Int may hold. If true, the Int may contain whole numbers from 0 to 4,294,967,296, if false it may contain whole numbers from -2,147,483,647 to 2,147,483,647. This property has a default value of false. If this property is set to true while the calling object has a value less than 0, the value of the object is set to 0, otherwise the value is unaffected.

public Int Value: Value is used to get the value of the Int. Usually used when creating extension methods.

2.2.2 Double

The type Double represents a floating point number, a decimal. A Double is a 64 bit floating point value. Double is a subclass of Int, so it inherits Ints operators, methods, and properties. I describe here only the operators, methods, and properties that are overrode or added in the Double class.

Initialization

An double can be initialized in any of the following ways.

```
Double a = 42.0;
Double b = new Double(42.69);
Double c = 47+12*2; // c #=> 71
```

Multiple Doubles can be initialized at once in the following ways.

```
Double a = 41, b = 42, c = 43.5; // a #=> 41.0, b #=> 42.0, c #=>
    43.5
Double d, e, f = 42; // d #=> 42.0, e #=> 42.0, f #=> 42.0
```

Binary Operators

There are several operators available to the Double class such as: `+`, `-`, `/`, `*`, `%`, `**`, `==`. These operators all function in the normal mathematical way, with the double equals representing equality checking. These operators be used between any two Doubles or other subclasses of Int.

The `+` operator arithmetically adds its two operands numeric value together and evaluates to the result. The `+` operator is a binary infix operator. For example, after the following code has executed, c contains 42.5.

```
Double a = 35;
Double b = 7.5;
Double c = a + b; // c #=> 42.5
```

The `-` operator arithmetically subtracts the numeric value of operand 2 from the numeric value of operand 1 and evaluates to the result. The `-` operator is a binary Double operators. For example, after the following code has executed, c contains 42.5.

```
Double a = 56.5;
Double b = 14;
Double c = a - b; // c #=> 42.5
```

The `*` operator arithmetically multiplies its two operands' numeric values together and evaluates to the result. The `*` operator is a binary infix operator. For example, after the following code has executed, c contains 42.

```
Double a = 7;
Double b = 6.4;
Double c = a * b; // c #=> 44.8
```

The `/` operator arithmetically divides the numeric value of operand 1 by the numeric value of operand 2 and evaluates to the result. The second operand may not be zero. The `/` operator is a binary infix operator. For example, after the following code has executed, c contains 42.

```
Double a = 126.0;
Double b = 3.0;
Double c = a / b; // c#=> 42
```

The return type of `/` is the type of its most precise operand. An Double divided by and `Double` is always and `Double`, and an `Int` divided by a `Double` evaluates to a `Double`.

```
Double a = 12, b = 8;
Double c = a/b; //c #=> 1.5

Double d = 8;
Double e = a/d; //e #=> 1.5
```

The `%` operator is the modulus or mod operator. It evaluates to the remainder of the numeric value of operand 1 divided by the numeric value of operand 2. The `%` operator is a binary infix operator. For example, after the following code has executed, c contains 2.5.

```
Double a = 14.5, b = 3;
Double c = a % b; // c #=> 2.5
```

The `**` operator raises the numeric value of operand 1 to the power of the numeric value of operand 2 and evaluates to the result. The `**` operator is a binary infix operator. For example, after the following code has executed, c contains 32.

```
Double a = 2, b = 5;
Double c = a**b; // #=> 32.0
```

Unary Operators

There are several unary operators available to the Double class such as ++, --. These unary operators work on a single Double.

The `--` operator decrements the numeric value of the operand. The `--` operator takes one argument in either of the following ways: operand`--` and `--`operand, which are post-decrement and pre-decrement, respectively.

```
Double a = 43.125;
a--; //a #=> 42.125
```

When post-decrementing, the value of the variable is used before it is decrementing, when pre-decrementing, the value of the variable is used after it is decrementing.

```
Double x,y = 42;
Double a = --x; // a #=> 41.0
Double b = x--; // b #=> 42.0
```

The ++ operator increments the numeric value of the operand. The ++ operator takes one argument in either of the following ways: operand++ and ++operand, which are post-increment and pre-increment, respectively.

```
Double a = 41;
a++; // a #=> 42.0
```

When post-incrementing the value of the variable is used before it is incremented, when pre-incrementing the value of the variable is used after it is incremented.

```
Double x,y = 42;
Double a = ++x; // a #=> 43.0
Double b = x++; // b #=> 42.0
```

Methods

These methods are available to the Double class, as well as all subclasses of Double due to inheritance. Methods generally start with a lowercase letter, continuing in camel case. They are called with the conventional dot (.) operator, with parentheses after the name. For example, the following code uses the .copy() method to put a copy of x into y.

```
Double x = 42;
Double y = x.copy(); //y = 42.0
```

public void copy() The .copy() method returns a new instance of the Double class that is the same as the calling object.

```
Double x = 42;
Double y = x.copy();
// at this point x and y have the same value
x++;
// at this point x = 43.0, y = 42.0
```

public Bool equals(Double i) The .equals(Double i) method returns a Bool indicating whether or not the calling object and the parameter have equal values. This method returns false if the parameter is null.

```
Double x, y = 42;
Bool a = x.equals(y); // a = true

Double z = 43;
Bool b = x.equals(z); // b = false

Double i;
Bool c = x.equals(i); //c = false
```

public String toString() The .toString() method returns a String containing the numeric value of the Double.

Properties

Doubles' properties are accessed with the dot (.) operator, similar to methods, with the difference being that properties never have parentheses at the end or parameters. Properties generally start with a capital letter and continue in camel case.

2.2.3 Float

The type Float represents a floating point number, a decimal. A Float is a 32 bit floating point value. Float is a subclass of Int, so it inherits Ints operators, methods, and properties. I describe here only the operators, methods, and properties that are overrode or added in the Float class.

Initialization

A Float can be initialized in any of the following ways.

```
Float a = 42.0;
Float b = new Float(42.69);
Float c = 47+12*2; // c #=> 71
```

Multiple Floats can be initialized at once in the following ways.

```
Float a = 41, b = 42, c = 43.5; // a #=> 41.0, b #=> 42.0, c #=>
    43.5
Float d, e, f = 42; // d #=> 42.0, e #=> 42.0, f #=> 42.0
```

Binary Operators

There are several operators available to the Float class such as: `+`, `-`, `/`, `*`, `%`, `**`, `==`. These operators all function in the normal mathematical way, with the double equals representing equality checking. These operators be used between any two Floats or other subclasses of Int.

The `+` operator arithmetically adds its two operands numeric value together and evaluates to the result. The `+` operator is a binary infix operator. For example, after the following code has executed, c contains 42.5.

```
Float a = 35;
Float b = 7.5;
Float c = a + b; // c #=> 42.5
```

The `-` operator arithmetically subtracts the numeric value of operand 2 from the numeric value of operand 1 and evaluates to the result. The `-` operator is a binary infix operator. For example, after the following code has executed, c contains 42.5.

```
Float a = 56.5;
Float b = 14;
Float c = a - b; // c #=> 42.5
```

The `*` operator arithmetically multiplies its two operands' numeric values together and evaluates to the result. The `*` operatoris a binary infix operator. For example, after the following code has executed, c contains 44.8.

```
Float a = 7;
Float b = 6.4;
Float c = a * b; // c #=> 44.8
```

The `/` operator arithmetically divides the numeric value of operand 1 by the numeric value of operand 2 and evaluates to the result. The second operand may not be zero. The `/` operator is a binary infix operator. For example, after the following code has executed, c contains 42.0.

```
Float a = 126.0;
Float b = 3.0;
Float c = a / b; // c #=> 42.0
```

The return type of / is the type of its most precise operand. A `Float` divided by a `Float` is always a `Float`, and an `Int` divided by a `Float` evaluates to a `Float`, and a `Float` divided by a `double` evaluates to a `Double`.

```
Float a = 12, b = 8;
Float c = a/b; //c #=> 1.5

Float d = 8;
Float e = a/d; //e #=> 1.5
```

The % operator is the modulus or mod operator. It returns the remainder of the numeric value of operand 1 divided by the numeric value of operand 2. The % operator is a binary infix operator. For example, after the following code has executed, c contains 2.5.

```
Float a = 14.5, b = 3;
Float c = a % b; // c #=> 2.5
```

The ** operator raises the numeric value of operand 1 to the power of the numeric value of operand 2 and evaluates to the result. The ** is a binary infix operator. For example, after the following code has executed, c contains 32.

```
Float a = 2, b = 5;
Float c = a**b; // #=> 32.0
```

Unary Operators

There are several unary operators available to the Float class such as ++, --. These unary operators work on a single Float.

The -- operator decrements the numeric value of parameter 1. The -- operator takes one argument in either of the following ways: operand-- and --operand, which are post-decrement and pre-decrement, respectively.

```
Float a = 43.125;
a--; //a #=> 42.125
```

When post-decrementing, the value of the variable is used before it is decrementing, when pre-decrementing, the value of the variable is used after it is decrementing.

```
Float x,y = 42;
Float a = --x; // a #=> 41.0
Float b = x--; // b #=> 42.0
```

The ++ operator increments the numeric value of the operand. The ++ operator takes one argument in either of the following ways: operand++ and ++operand, which are post-increment and pre-increment, respectively.

```
Float a = 41;
a++; // a #=> 42.0
```

When post-incrementing the value of the variable is used before it is incremented, when pre-incrementing the value of the variable is used after it is incremented.

```
Float x,y = 42;
Float a = ++x; // a #=> 43.0
Float b = x++; // b #=> 42.0
```

Methods

These methods are available to the Float class, as well as all subclasses of Float due to inheritance. Methods generally start with a lowercase letter, continuing in camel case. They are called with the conventional dot (.) operator, with parentheses after the name. For example, the following code uses the .copy() method to put a copy of x into y.

```
Float x = 42;
Float y = x.copy(); //y = 42.0
```

public void copy()

The .copy() method returns a new instance of the Float class that is the exact same as the calling object.

```
Float x = 42;
Float y = x.copy();
// at this point x and y have the same value
x++;
// at this point x = 43.0, y = 42.0
```

public Bool equals(Float i)

The .equals(Float i) method returns a Bool indicating whether or not the calling object and the parameter have equal values. This method returns false if the parameter is null.

```
Float x, y = 42;
Bool a = x.equals(y); // a = true

Float z = 43;
Bool b = x.equals(z); // b = false

Float i;
Bool c = x.equals(i); //c = false
```

public String toString() The .toString() method returns a String containing the numeric value of the Float.

Properties

Floats' properties are accessed with the dot (.) operator, similar to methods, with the difference being that properties never have parentheses at the end or parameters. Properties generally start with a capital letter and continue in camel case.

2.2.4 Long

The type Long represents an Integer, a whole number. A Long is a 64 bit value and by default can store any value from -9.223372e+18 to 9.223372e+18. Long is a subclass of Int, so it inherits Ints operators, methods, and properties. I describe here only the operators, methods, and properties that are overrode or added in the Long class.

Initialization

An Long can be initialized in any of the following ways.

```
Long a = 42;
Long b = new Long(42);
Long c = 47+12*2; // c #=> 71
```

Multiple Longs can be initialized at once in the following ways.

```
Long a = 41, b = 42, c = 43; // a #=> 41, b #=> 42, c #=> 43
Long d, e, f = 42; // d #=> 42, e #=> 42, f #=> 42
```

Binary Operators

There are several operators available to the Long class such as: `+`, `-`, `/`, `*`, `%`, `==`. These operators all function in the normal mathematical way, with the double equals representing equality checking. These operators be used between any two Longs or subclasses of Int.

The `+` operator arithmetically adds its two operand numeric value together and evaluates to the result. The `+` operator is a binary infix operator. For example, after the following code has executed, c contains 42.

```
Long a = 35;
Long b = 7;
Long c = a + b; // c #=> 42
```

The `-` operator arithmetically subtracts the numeric value of operand 2 from the numeric value of operand 1 and evaluates to the result. The `-` operator is a binary infix operator. For example, after the following code has executed, c contains 42.

```
Long a = 56;
Long b = 14;
Long c = a - b; // c #=> 42
```

The * operator arithmetically multiplies its two operands' numeric values together and evaluates to the result. The * operator is a binary infix operator. For example, after the following code has executed, c contains 42.

```
Long a = 7;
Long b = 6;
Long c = a * b; // c #=> 42
```

The / operator arithmetically divides the numeric value of operand 1 by the numeric value of operand 2 and evaluates to the result. The / operator is a binary infix operator. For example, after the following code has executed, c contains 42.

```
Long a = 126;
Long b = 3;
Long c = a / b; // c #=> 42
```

The return type of / is the type of its most precise parameter. An Long divided by and Long is always and Long, however and Long divided by a Double evaluates to a Double.

```
Long a = 12, b = 8;
Double c = a/b; //c #=> 1

Double d = 8;
Double e = a/d; //e #=> 1.5
```

The % operator is the modulus or mod operator. It returns the remainder of the numeric value of operand 1 divided by the numeric value of operand 2. The % operator is a binary infix operator. For example, after the following code has executed, c contains 2.

```
Long a = 14, b = 3;
Long c = a % b; // c #=> 2
```

The ** operator raises the numeric value of operand 1 to the power of the numeric value of operand 2 and returns the result. The ** takes two is a binary infix operator. For example, after the following code has executed, c contains 32.

```
Long a = 2, b = 5;
Long c = a**b; // #=> 32
```

Unary Operators

There are several unary operators available to the Long class such as ++, --. These unary operators work on a single Long.

The -- operator decrements the numeric value of the operand. The -- operator takes one argument in either of the following ways: operand-- and --operand, which are post-decrement and pre-decrement, respectively.

```
Long a = 43;
a--; //a #=> 42
```

When post-decrementing, the value of the variable is used before it is decrementing, when pre-decrementing, the value of the variable is used after it is decrementing.

```
Long x,y = 42;
Long a = --x; // a #=> 41
Long b = x--; // b #=> 42
```

The ++ operator increments the numeric value of the operand. The ++ operator takes one argument in either of the following ways: operand++ and ++operand, which are post-increment and pre-increment, respectively.

```
Long a = 41;
a++; // a #=> 42
```

When post-incrementing the value of the variable is used before it is incremented, when pre-incrementing the value of the variable is used after it is incremented.

```
Long x,y = 42;
Long a = ++x; // a #=> 43
Long b = x++; // b #=> 42
```

Methods

These methods are available to the Long class, as well as all subclasses of Long due to inheritance. Methods generally start with a lowercase letter, continuing in camel case. They are called with the conventional dot (.) operator, with parentheses after the name. For example, the following code uses the .copy() method to put a copy of x into y.

```
Long x = 42;
Long y = x.copy();
```

public void copy()

The .copy() method returns a new instance of the Long class that is the exact same as the calling object.

```
Long x = 42;
Long y = x.copy();
// at this point x and y have the same value
x++;
// at this point x = 43, y = 42
```

public Bool equals(Long i)

The .equals(Long i) method returns a Bool indicating whether or not the calling object and the parameter have equal values. This method returns false if the parameter is null.

```
Long x, y = 42;
Bool a = x.equals(y); // a = true

Long z = 43;
Bool b = x.equals(z); // b = false

Long i;
Bool c = x.equals(i); //c = false
```

public String toString() The .toString() method returns a String containing the numeric value of the Long.

Properties

Longs' properties are accessed with the dot (.) operator, similar to methods, with the difference being that properties never have parentheses at the end or parameters. Properties generally start with a capital letter and continue in camel case.

public Bool PositiveOnly: PositiveOnly is used to set he range of values a particular Long may hold. If true, the Long may contain whole numbers from 0 to 1.8446744e+19, if false it may contain whole numbers from -9.223372e+18 to 9.223372e+18. This property has a default value of false. If this property is set to true while the calling object has a value less than 0, the value of the object is set to 0, otherwise the value is unaffected.

2.2.5 Char

The type Char represents a single ASCII character.

Initialization

An Char can be initialized in any of the following ways.

```
Char a = 'a';
Char b = new Char('%');
Char c = 77; // c #=> M
```

Multiple Chars can be initialized at once in the following ways. A character initialized with a number will be assigned the appropriate character based on the numeric values in the ASCII table.

```
Char a = 'a', b = 'f', c = 103; // a #=> a, b #=> f, c #=> g
Char d, e, f = 'A'; // d #=> A, e #=> A, f #=> A
```

Binary Operators

There are several operators available to the Char class such as: `+`, `-`, `/`, `*`, `%` `**`, `==`. These operators all function in the normal mathematical way, with the double equals representing equality checking. These operators be used between any two Chars or a Char and and Int, or subclass of Int. When using mathematical operators on a Char, the numeric value of the Char based on the ASCII table is used.

The `+` operator arithmetically adds its two operands numeric value together and evaluates to the result. The `+` operator is a binary infix operator. For example, after the following code code has executed, c contains the character 'h'.

```
Char a = 'a';
Char b = 7;
Char c = a + b; // c #=> h
```

The `-` operator arithmetically subtracts the numeric value of operand 2 from the numeric value of operand 1 and evaluates to the result. The `-` operatoris a binary infix operator. For example, after te following code has executed, c contains the character 'b'.

```
Char a = 'd';
Char b = 2;
Char c = a - b; // c #=> b
```

The `*` operator arithmetically multiplies its two operands' numeric values together and evaluates to the result. The `*` operator is a binary infix operator. For example, after the following code has executed, c contains the character 'F'.

```
Char a = 7;
Char b = 10;
Char c = a * b; // c #=> F
```

The `/` operator arithmetically divides the numeric value of operand 1 by the numeric value of operand 2 and evaluates to the result. The `/` operatoris a binary infix operator. For examples, after the following code has executed, c contains the character '('.

```
Char a = 'z';
Char b = 3;
Char c = a / b; // c #=> (
```

The % operator is the modulus or mod operator. It evaluates to the remainder of the numeric value of operand 1 divided by the numeric value of operand 2. The % operator is a binary infix operator. For example, after the following code has executed, c contains the character '1'.

```
Char a = 'p';
char b = '?';
Char c = a % b; // c #=> 1
```

The ** operator raises the numeric value of operand 1 to the power of the numeric value of operand 2 and evaluates to the result. The ** is a binary infix operator. For example, after the following code has executed, c contains the character '@'.

```
Char a = 2, b = 6;
Char c = a**b; // c #=> @
```

Unary Operators

There are several unary operators available to the Char class such as ++, --. These unary operators work on a single Char.

The -- operator decrements the numeric value of the operand. The -- operator takes one argument in either of the following ways: operand-- and --operand, which are post-decrement and pre-decrement, respectively.

```
Char a = 'v';
a--; //a #=> u
```

When post-decrementing, the value of the variable is used before it is decrementing, when pre-decrementing, the value of the variable is used after it is decrementing.

```
Char x,y = B;
Char a = --x; // a #=> A
Char b = x--; // b #=> B
```

The ++ operator increments the numeric value of the operand. The ++ operator takes one argument in either of the following ways: operand++ and ++operand, which are post-increment and pre-increment, respectively.

```
Char a = 'a';
a++; // a #=> b
```

When post-incrementing the value of the variable is used before it is incremented, when pre-incrementing the value of the variable is used after it is incremented.

```
Char x,y = 'F';
Char a = ++x; // a #=> G
Char b = x++; // b #=> F
```

Methods

The below methods are available to the Char class, as well as all subclasses of Char. Methods generally start with a lowercase letter, continuing in camel-case. They are called with the conventional . operator, with parentheses after the name. For example, the following code uses the .copy() method to put a copy of x into y.

```
Char x = 42;
Char y = x.copy(); //y #=> 42
```

public void copy()

The .copy() method returns a new instance of the Char class that is the exact same as the calling object.

```
Char x = 42;
Char y = x.copy();
// at this point x and y have the same value
x++;
// at this point x #=> 43, y #=> 42
```

public Bool equals(Char i)

The .equals(Char i) method returns a Bool indicating whether or not the calling object and the parameter have equal values. This method returns false if the parameter is null.

```
Char x, y = 42;
Bool a = x.equals(y); // a = true

Char z = 43;
Bool b = x.equals(z); // b = false

Char i;
Bool c = x.equals(i); //c = false
```

public Int toInt()

The .toInt() method returns the numeric ASCII value of the character.

public String toString()

The toString method returns a String containing a single character representing the Char's value.

Properties

Chars' properties are accessed with the dot (.) operator, similar to methods, with the difference being that properties never have parentheses at the end or parameters. Properties generally start with a capital letter and continue in camel case.

2.2.6 String

The type String represents a sequence of characters such as `"abc"` or `"hello there"`.

Initialization

Strings can be initialized in any of the following ways.

```
String s = "hello world";
String a = new String("goodbye");
```

Multiple Strings can be initialized at once in the following ways.

```
String a = "Hello", b = "world", c = "Goodbye";
String d, e, f = "Hello world";
```

Operators

The String class provides several operators such as: `+, -, *, <, >, ==, []`. These operators may be used between any two Strings or a String and any other object that has operators defined for working with Strings, such as the Char class. In these cases the Char is converted to a String via the toString() method before the operation is performed.

The `+` operator is the concatenation operator. It evaluates to a new string containing operand 1 concatenated with operand 2. The `+` operator is a binary infix operator. For example, after the following code has executed, c contains "Hello World".

```
String a = "Hello ";
String b = "world!";
String c = a + b; // c #=> "Hello World";
```

The `<` operator compares strings alphabetically. This operator evaluates to true if operand 1 comes alphabetically before operad 2, and evaluates to false otherwise The `<` operator is a binary infix operator. For example, after the following code has executed, bool contains true and b contains false.

```
String a = "abc";
String b = "abd";
Bool bool = a < b; // bool #=> true
bool = b < a; // bool #=> false
```

Capital letters appear earlier in the ASCII table than lower case letters.

```
String a = "abc";
String b = "XYZ";
Bool bool = a < b; // bool #=> false
bool = b < a; // bool #=> true
```

The > operator compares Strings alphabetically. This operator evaluates to true if operand 1 comes alphabetically after operand 2, and evaluates to false otherwise. The > operator is a binary infix operator. For example, after the following code has executed, bool contains false and b contains true.

```
String a = "abc";
String b = "abd";
Bool bool = a > b; // bool #=> false
bool = b > a; // bool #=> true
```

Capital letters appear earlier in the ASCII table than lower case letters.

```
String a = "abc";
String b = "XYZ";
Bool bool = a > b; // bool #=> true
bool = b > a; // bool #=> false
```

The == operator checks for equality between two strings. If the operands are exactly the same, it evaluates to true. Otherwise it evaluates to false. The == operator is a binary infix operator.

```
String a = "hello", b = "goodbye", c = "Hello", d = "goodbye";
Bool bool = a == b; // bool #=> false
bool = a == c; // bool #=> false
bool = b == d; // bool #=> true
```

The - operator is overloaded and accepts two parameters. If used with a String s and an integer n, it returns s with n characters removed from the right-hand side of the string.

```
String s = "Hello World!";
String b = s-4; // b #=> "Hello Wo"
```

If used with two Strings s and r, it returns s minus all instances of r.

```
String s = "Goodbye!";
String r = "o";
String a = s-r; // a #=> "Gdbye!"
a = s-"odb"; // a #=> "Goye!"

s = "Test ttt";
a = s - "tt"; // a #=> Test t
```

The * operator takes a String as operand 1 and an integer as operand 2. It evaluates to a String containing n copies of operand 1, where n = operand 2.

```
String a = "abc";
String b = a*3; // b #=> "abcabcabc";
```

The [] operator takes a String as parameter 1 and an integer as parameter 2 in this way: parameter1[parameter 2]. This operator returns the character at index parameter2. This operator is an alias of charAt method described below.

```
String str = "Hello World!";
Char c = str[0]; // c #=> H
c = str[4]; // c #=> o
//str[4] is equivalent to str.charAt(4)
```

Methods

These methods are available to the String class, as well as all subclasses of String. Methods generally start with a lowercase letter, continuing in camel-case. They are called with the conventional dot (.) operator, with parentheses after the name. For example, the following code uses the .copy() method to put a copy of x into y. The collection methods map, reduce, and select are also available to String, for more information see the collections reference.

public void copy()

The .copy() method returns a new instance of the String class that is the exact same as the calling object.

```
String x = "Hello";
String y = x.copy();
// at this point x and y have the same value
```

public Bool equals(String s)

The .equals(String s) method returns a Bool indicating whether or not the calling object and the parameter have equal values. This method returns false if the parameter is null.

```
String x, y = "hello";
Bool a = x.equals(y); // a = true

String z = "world";
Bool b = x.equals(z); // b = false

String s;
Bool c = x.equals(i); //c = false
```

public String[] split(String delimiter) The .split(String delimiter) method returns an array containing the contents of the String, split at every instance of the delimiter.

```
String s = "hello,there,world";

String[] arr = s.split(",");
//arr #=> ["hello","there","world"];
```

public String[] split(Char delimiter) The .split(Char delimiter) method returns an array containing the contents of the String, split at every instance of the delimiter.

```
String s = "hello,there,world";

String[] arr = s.split(',');
//arr #=> ["hello","there","world"];
```

public String snbString(Int b, Int e)

The .subString(Int b, Inte) method returns a String that is a subsection of the original string from the character at index b and up to but not including the character at index e. The first character is index zero.

```
String a = "Hello World!";
String b = a.subString(1,5); // b #=> ello
b = a.subString(1,9); // a #=> ello Wor
```

public void subString!(Int b, Int e)

The destructive .subString(Int b, Int e) method keeps a subsection of the calling String from the character at index b and up to but not including the character at index e. The first character is index zero.

```
String a = "Hello World!";
a.subString!(1,5); // a #=> ello
a = "Hello World!";
a.subString!(1,9); // a #=> ello Wor
```

public Char charAt(Int i)

The .charAt(Int i) method returns a Char, the character at the specified index. The first character is index zero.

```
String str = "Hello World!";
Char c = str.charAt(4); // c #=> o
c = str.charAt(0) // c 3 #=> H
```

public Int length()

The .length() method returns an Int representing the number of characters in the String.

```
String str = "Hello World!";
Int i = str.length(); // i #=> 12
i = "abc".length(); // i #=> 3
```

public String concat(String str)

The .concat(String str) method returns a String that is the calling String concatenated with the parameter String.

```
String str = "Hello";
String c = " World!";
String r = str.concat(c); // r #=> Hello World!
r = "String 1".concat("String 2"); // r #=> String 1String 2
```

public void concat!(String str)

The destructive .concat(String str) method concatenates the parameter String onto the end of the calling String.

```
String str = "Hello";
String c = " World!";
str.concat!(c); // str #=> Hello World!
r = "String 1" + "String 2"; // r #=> String 1String 2
```

public Bool contains(String str)

The .contains(String str) method returns true if the calling String contains an instance of the parameter String.

```
String str = "Some example text";
String s = "Some";
Bool bool = str.contains(s); // bool #=> true
bool = str.contains("Some text"); // bool #=> false
bool = str.contains("t"); // bool #=> true
```

public Bool contains(Char c)

The .contains(Char c) method returns true if the calling String contains an instance of the parameter Character.

```
String str = "Some example text";
Char ch = 'S';
Bool bool = str.contains(ch); // bool #=> true
bool = str.contains('E'); // bool #=> false
bool = str.contains('t'); // bool #=> true
```

public Int indexOf(String str)

The .indexOf(String str) method returns an Int, the index of the first character of the parameter String in the calling String. It returns null if it does not exist.

```
String str = "Hello World!";
Int i = str.indexOf("Hello"); // i #=> 0
i = str.indexOf("Wor"); // i #=> 6
i = str.indexOf("Word"); // i #=> null
```

public Int indexOf(Char c)

The .indexOf(Char c) method returns an Int, the index of the parameter character in the calling String.

```
String str = "Hello World!";
Char c = 'W';
Int i = str.indexOf(c); // i #=> 6
i = str.indexOf('a'); // i #=> null
```

public Bool isEmpty()

The .isEmpty() method returns a Bool, true if the string contains no characters, false if it contains at least one character.

```
Sting str = "%";
Bool b = str.isEmpty(); // b #=> false
str = "";
b = str.isEmpty(); // b #=> true
```

public String replace(Char oldChar, Char newChar)

The .replace(Char oldChar, char newChar) method returns a string where all instances of the Char oldChar in the String have been replaced with the Char newChar.

```
String str = "Hello World!";
String r = str.replace('l', 'w'); // r #=> Hewwo Worwd!
r = str.replace(' ','_'); // r #=> Hello_World!
```

public void replace!(Char oldChar, Char newChar)

The destructive .replace(Char oldChar, Char newChar) method replaces all instances of the Char oldChar in the String with the Char newChar.

```
String str = "Hello World!";
str.replace('l', 'w'); // str #=> Hewwo Worwd!
```

public String replace(Striug oldString, Striug newString)

The .replace(String oldString, String newString) method returns a string where all instances of the String oldString in the String have been replaces with the String newString.

```
String str = "This is goodbye!!";
String r = str.replace("is","watermelon");
   // r #=> Thwatermelon watermelon goodbye!!
```

public void replace!(String oldString, String newString)

The destructive .replace(String oldString, String newString) method replaces all instances of oldString in the String with newString.

```
String str = "This is goodbye!!";
str.replace!("is", "watermelon");
   // str #=> Thwatermelon watermelon goodbye!!
str.replace("watermelon", ""); // str #=> Th  goodbye!!
```

public String trim()

The .trim() method returns the calling String with all of the leading and trailing whitespace removed.

```
String str = " Hello   World      ";
String r = str.trim(); // r #=> Hello   World
```

public void trim!()

The destructive .trim() method removes all leading and trailing whitespace from the calling String.

```
String str = " Hello   World      ";
str.trim!(); // str #=> Hello   World
```

public String toUpper()

The .toUpper() method returns a String where all of the lowercase alphabet characters in the calling String have been uppercased. All other characters are unchanged.

```
String str = "Hello World!";
String r = str.toUpper(); // r #=> HELLO WORLD!
```

public void toUpper!()

The destructive .toUpper() method uppercases all lowercase alphabet characters in the calling String. All other characters are unchanged.

```
String str = "Hello World!";
str.toUpper!(); // str #=> HELLO WORLD!
```

public String toLower()

The .toLower() method returns a String where all of the uppercase alphabet characters in the calling String have been lowercased. All other characters are unchanged.

```
String str = "Hello World!";
String r = str.toLower(); // r #=> hello world!
```

public void toLower!()

The destructive .toLower() method lowercases all uppercase alphabet characters in the calling String. All other characters are unchanged.

```
String str = "Hello World!";
str.toLower!(); // str #=> hello world!
```

Properties

Strings' properties are accessed with the dot (.) operator, similar to methods, with the difference being that properties never have parentheses at the end or parameters. Properties generally start with a capital letter and continue in camel case.

public Bool UpperCaseOnly: If UpperCaseOnly is true all lowercase characters in the String will be uppercased. UpperCaseOnly is false by default.

public Bool LowerCaseOnly: If LowerCaseOnly is true all uppercase characters in the String will be lowercased. LowerCaseOnly is false by default.

2.2.7 Bool

The Bool type represents a binary value, either true or false;

Initialization

Bools can be initialized in any of the following ways.

```
Bool a = true;
Bool b = 1 > 4; // b #=> false
```

Multiple Bools may be initialized in the following ways.

```
Bool a, b, c = false; // a, b, and c are false
Bool d = true, e = false, f = false;
```

Operators

There are several operators available to the Bool class such as { !, ==, &&, ||}. These operators may be used between any two Bools or expressions that evaluate to a Bool.

The `!` operator evaluates to the inverse of the calling Bool.

```
Bool b = false;
Bool r = !b; //r #=> true
r = !r; // r #=> false
```

The binary AND operator (`&&`) evaluates to true if both it's operands are true, and evaluates to false otherwise.

```
Bool a = true, b = true, c = false;
Bool r = a && b; // r #=> true
r = a && c; // r #=> false
```

The binary OR operator (`||`) evaluates to true if either of it's operands are true, and evaluates to false if both operands are false.

```
Bool a = true, b = true, c = false, d = false;
Bool r = a || b; // r #=> true
r = a || c; // r #=> true
r = c || d; // r #=> false
```

The `==` is the equality operator. It is a binary infix operator. It evaluates to true if it is two operands have the same value, and evaluates to false otherwise

```
String a, b = false;
String c = a == b; // c #=> true
String d = d == a; // d #=> false
```

Methods

These methods are available to the Bool class, as well as all subclasses of Bool. Methods generally start with a lowercase letter, continuing in camel-case. They are called with the conventional dot (.) operator, with parentheses after the name. For example, the following code uses the .copy() method to put a copy of x into y.

```
Bool x = true;
Bool y = b.copy(); // y #=> true
```

public void toString() The .toString() method returns a String based on the value of the Bool. It returns "true" if the Bool is true, "false" if it is false.

```
Bool b = true;
String str = b.toString(); // str #=> true
b = false;
str = b.toString(); // str #=> false
```

2.3 The Object Class

The Object class is the base for all other classes. Each of the built in types extend the object class, and all user-created classes implicitly extend the Object class. This means that all user created objects inherit the basic methods and properties provided by the object class. Some of these may be overridden by the sub-objects, but they will always exist.

2.3.1 Methods

The following methods are available to all objects, and perform the following actions unless stated otherwise in the documentation for that sub-class.

public String toString()

The toString method returns a String containing the memory address of the Object. This is meant to be overridden by sub-classes, but in the case that it is not, it allows reference equality checks.

```
Object o1 = new Object();
Object o2 = o1.ref;
Object o3 = new Object();

String str = o1.toString(); // str #=> 2F45A92D
```

```

Bool b = o1.toString() == o2.toString(); // b #=> true
b = o1.toString() == o3.toString(); // b #=> false
```

2.3.2 Properties

The following properties are available to all objects, and perform the following actions unless stated otherwise in the documentation for that class.

public Object Ref:

This property is read-only and contains a reference to the calling object. It allows multiple references to the same object. The following example will use the Int class, but because all objects extend the Object class it will work in the exact same way with any object.

```
Int i = 10;
Int b = i.Ref; // b #=> 10
b = 12;
// at this point i and b equal 12
i--;
// at this point b and i equal 11
```

public String Type

The Type property is read-only and contains the type of the calling object. It can be used to check if two objects have the same type.

```
Int i = 10;
String str = "10";
Bool b = i.Type == str.Type; // b #=> false
b = i.toString().Type == str.Type; // b #=> true
b = str.Type == Int; // b #=> false
```

2.4 User Defined Objects

Below is the documentation for creating your own objects.

2.4.1 Access Modifiers

Access Modifiers are how we set who can access certain things. Access modifiers can be used with Fields, Classes, Methods, etc. There are three access modifiers, which each correspond to an access level. If the access modifier in a declaration is left empty, it defaults to protected. The access modifiers are:

- `public` - Allows access to the world
- `protected` - Allows access within the containing class or subclasses
- `private` - Allows access within the containing class

2.4.2 Object Declaration

Class declaration consists of <access modifier>class <identifier><block>. The following class declaration is for a public class named Book, with an empty code block.

```
public class Book{

}
```

This example depicts the simplest form of a class declaration. In order to use this object we need to be able to create one, or 'new' one. We have not added a constructor yet, but when a class lacks a constructor a default constructor that is automatically generated. The default constructor has the following structure: public void <class identifier>(). To use this constructor we use the `new` keyword followed by the constructor name.

```
Book b = new Book();
```

2.4.3 Creating Properties

A property is just a variable that is stored in an object. A property's accessibility can be altered with the use of access modifiers such as `public` and **private** keywords. To declare a field, simply declare a variable in the class's code block. Properties conventionally start with an uppercase letter and continue in camel-case. The following example adds a title property to our book object.

```
public class Book{
   public String Title = "Programming for Dummies";
}
```

To access fields you use the dot (.) operator similar to how you access Properties.

```
Book b = new Book();
String str = b.Title; // str #=> Programming for Dummies
b.Title = "The Biography of Michael Stallman";
str = b.Title; // str #=> The Biography of Michael Stallman
```

Objects may have as many fields as necessary provided they are uniquely named.

2.4.4 Creating a Constructor

A user-defined constructor accepts any information needed and set up the object for use. In almost all cases we will want to create our own constructor. For example, in our Book class we are currently setting the title of the book when we declare the variable. This is something that will usually be done in the constructor. A constructor's accessibility can be altered with the use of access modifiers. (e.g. **public** and **private** keywords).

```
public class Book{
   String Title;
   public Book(){
      Title = "Programming for Dummies";
   }
}
```

This is functionally no different than our previous example, but we can now build on this to create books with different titles. Up until now, no matter how many books we create, they all have the same name. This is undesired behavior, and we would rather be able to create Book objects with different titles from the same class. To do this we need to pass a parameter containing our desired book title to the constructor. Constructors accept parameters in the same way methods do.

```java
public class Book{
   String Title;
   public Book(String t){
      Title = t;
   }
}
```

We may now create books with different titles:

```java
Book a = new Book("The Biography of Michael Stallman");
Book b = new Book("Programming for Dummies");
// Note that we no longer have a default constructor, therefore
// we cannot do Book c = new Book(); anymore, we must provide
// a String parameter
```

2.4.5 Adding Methods to Objects

Whereas Properties add memory to a Class, methods add behavior. In general, most interaction with an object will be through method calls. A function's accessibility can be altered with the use of access modifiers such as `public` and `private` keywords. Method declaration consists of <access modifier><return type><name>(<parameter list>)<code block >. In the following example we create a public method that returns a String and accepts no parameters.

```java
public class Book{
   String Title;
   public Book(String t){
      Title = t;
   }

   public String getTitle(){
      return Title;
   }
}
```

Methods are called with the dot (.) operator on an instance of an object.

```
Book b = new Book("Tales of Arkansas");
String str = b.getTitle();// str #=> Tales of Arkansas
```

Methods that return nothing use the return type `void`.

```
public class Book{
   String Title;
   public Book(String t){
      Title = t;
   }

   public void setTitle(String t){
      Title = t;
   }
}
```

2.4.6 Overloading Operators

Overloading operators allows user created objects to be compared or acted on via the standard operators, as well as allowing users to redefine how an operator effects or acts on an existing object. This can be done with similar syntax to a method: <access modifier><return type><operator><object type><code block>.

```
public class Book{
   String Title;
   Int Rating;
   public Book(String t){
      Title = t;
      Rating = 0;
   }

   //implements the < operator based on the alphebetic order of the
     books titles.
   public Bool < Book b2 {
      return Title < b2.Title
   }

   //implements the -= operator to decrement the rating of a book
   public void -= Int i{
        Rating = Rating - i;
    }
}
```

2.5 Extending Existing Objects

Existing objects may be extended in order to add functionality to an object, or to take advantage of functionality that already exists in an object while creating a more specific class by adding Properties and methods.

2.5.1 Inheriting from Another Object

Inheriting from another object allows the subclass to access any of its parent class's methods and fields that are not marked private. Sub-classing is also referred to as extending a class. To inherit from an object, the **extends** keyword is used. In the following example we have our super-class Employee and sub-class Manager.

```
public class Employee{
    String Name;
    public Employee(String n){
        Name = n;
    }
}

public class Manager extends Employee{

}
```

Available Methods

Any method in the super-class that is not marked private is directly available to the sub-class. The sub-class also has access to the super-class's constructors through the `super()` function. In the following example the sub-class uses the super-class's constructor, to avoid repeating code.

```
public class Employee{
    String name;
    public Employee(String n){
        name = n;
    }

    public String getName(){
        return name;
    }
}
public class Manager extends Employee{
    public Manager(String n){
        super(n); // calls Employee's constructor
    }
}
```

In the following example we use the two classes above. In it you can see Manager using one of Employees methods.

```
Manager m = new Manager("Steve Hennagin");
String str = m.getName(); // using one of Employee's methods
```

Overriding Methods

While sub-classes have access to their super-class's methods, they also have the ability to override them. This means new functionality can be assigned to a sub-class with a method that has the same signature as a method in it is super class.

```
public class Employee{
    String name;
    public Employee(String n){
        name = n;
    }

    public String getName(){
        return name;
    }
}
public class Manager extends Employee{
    public Manager(String n){
        super(n); // calls Employee's constructor
    }

    //Overriding the getName method in Employee
    public String getName(){
        return name + ", Manager";
    }
}
```

2.5.2 Extension Methods

Extension Methods allow you to add functionality to an object without extending the whole object. Extension Methods can be created for any object, including the built in objects. Extension Method declaration looks like <access modifier><return type><method name><parameter list>**extends** <class to be extended>. The **this** keyword is used to access the calling object. In the following example we add an extension method to the Int class to allow us to square an Int.

```
public Int square() extends Int{
    return this.Value*this.Value;
}

Int i = 4;
Int s = i.square(); // s #=> 16
```

Extension Methods may also be used to override methods that already exist in a class.

```
public class Employee{
    String firstName, lastName;
    public Employee(String fName, String lName){
        firstName = fName;
        lastName = lName;
    }

    public String getName(){
        return firstName;
    }
}

```

```
//extension method for Employee
public String getName() extends Employee{
   return this.firstName + " " + this.lastName;
}
```

2.6 Control Structures

Control structures provide a mechanism for non-sequential access of instructions. For example, an if statement is a control structure that allows for a specific set of instructions to be chosen based on the results of a test.

2.6.1 If Statements

An if statement executes code based on it's boolean condition. If the condition is true, it executes a section of code; if it is false it does not. The condition of an if statement must evaluate to a boolean value. The code portion can contain any other code.

```
Bool a = true;
Int x = 7;
if(a){
   x = 42;
}
//at this point x = 42

if(x>100){
   x = -1;
}

//at this point x = 42
```

2.6.2 If Else Statement

Sometimes you want to execute a certain piece of code when some condition is true, and a completely separate piece of code when the condition is false. This is what an if-else statement is for. An if-else statement is a normal if statement followed by an else statement.

```
Bool x = 42;
String str;
if(x < 100){
   str = "hello";
} else{
   str = "world";
}
//at this point str = hello

if(x < 100){
   str = "hello";
} else{
   str = "world";
}
//at this point str = world
```

2.6.3 Else If Block

The else-if block is used when you have multiple related conditions corresponding with multiple pieces of code. An else-if block is an if-else block followed by another if. This pattern may be continued indefinitely.

```
Int x = 42;
String str;
if(x == 10){
   str = "Logan";
} else if(x > 30){
   str = "Spencer";
} else{
   str = "Nolan";
}
//at this point str = Spencer

if(x > 9000){
   str = "Porcupine";
} else if(str == "Spencer"){
   x = -1;
}
//at this point x = -1
```

2.6.4 Switch Statement

A switch statement is used in similar cases to if-else-if blocks. It allows only one expression's value to be compared. A switch statement may also have a default case. A switch statement compares a value with a set of when clauses, executing the code after that when and stopping at the next when or the end of the switch statement, whichever comes first. Switch statements have no fall-through. The conditions of a switch statement may contain logical `&&` or `||` operators.

```
Int i = 42;
String str;
switch(i){
   when 7:
      str = "hello";
   when 18 || 0:
      str = "goodbye";
   default:
      str = "I didn't see you";
}
//at this point str #=> I didn't see you
```

The `<` and `>` operators may be used in a switch statement as well.

```
Int i = 42;
Char c;
switch(i){
   when >50:
      c = 'A';
   when <=42:
      c = 'B';
   default:
      c = 'Z';
}
//at this point c #=> B
```

2.6.5 Loops

A loop is a way of executing a piece of code multiple times. There are three types of loops in Emerald: While, do-while, and foreach. Emerald lacks the typical for loop found in most languages due to the enhanced functionality of the foreach loop.

While Loops

A while loop is a top tested loop that is used when a piece of code needs to be executed repeatedly while a certain condition is true. The condition for a while loop can be anything that evaluates to a Bool. The code inside of a while loop will never execute if the condition is false initially.

```
Int i = 0;
while(i<10){
   i = i+1;
}
//this loop executes 10 times and at this point i = 10

i = 0;
String str = "start";
while(i > 1){
   str = "end";
   i = i+1;
}
//the code inside this loop never executes and str = start
```

The boolean operators may also be used in loop conditions.

```
Int i = 42, j = 100;
while(i == 42 && j >= 0){
   j--;
}
```

Do While Loops

A do-while loop is a bottom tested loop that is used when a piece of code needs to be executed repeatedly while a certain condition is true. The condition for a do-while loop can be anything that evaluates to a Bool. The code in the body of a do-while loop will always execute at least once, even if the condition is false to begin with.

```
Int i = 0;
do{
   i--;
} while(i > 0)
// at this point i = -1
```

The boolean operators may also be used in loop conditions.

```
Int i = 0;
Bool b = false;
do{
   i++;
} while(i < 100 || b)
//this loop executes 100 times
```

Foreach Loops

A foreach loop executes a piece of code a certain number of times, or iterates through a collection. The body of a foreach loop may not be executed if the collection has a size of zero. A foreach loop has the following structure: foreach(<iterating variable> in <collection><code block>).

```
Int[] arr = {42,99,102,87};
Int sum = 0;
foreach(Int i in arr){
   sum = sum + i;
}
// at this point sum = 330
```

A foreach loop can also be used in a way similar to a traditional for loop, with indexes. This is possible because when an `Int` is used in this context it evaluates to a collection of all integers from 0 to the Int.

```
Int x = 8;
String str = "";
foreach(Int i in x){
   str = str + i + " ";
}
//at this point str = 0 1 2 3 4 5 6 7
```

A foreach loop may also be used to iterate to a negative number.

```
Int x = -10;
String str = "";
foreach(Int i in x){
   str = str + i + " ";
}
//at this point str = 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
```

A foreach loop may also be used to iterate down to 0.

```
Int x = 10;
String str = "";
foreach(Int i ni x){
   str = str + i + " ";
}
//at this point str = 10 9 8 7 6 5 4 3 2 1 0
```

2.7 Lambdas

A Lambda is an anonymous function, and can be written in-line or assigned to a variable. Lambda variables do not do strict type checking, and are therefore more prone to runtime errors due to incompatible types. Because strong type checking is not done, however, any type that supports the operations the lambda will perform on it may be passed into a lambda.

```
//a single line lambda that returns two times the value passed into
    it.
Lambda l = (Int x) => return 2*x;

//a single line lambda that accepts two numbers and returns the
   product of them
Lambda l = (Int x, Int y) => return x*y;

//a multiline lambda that returns the larger of the two values
   passed to it
Lambda higher = (Int x, Int y) => {
   if(x>y){
      return x;
   } else{
      return y;
    }
}
```

2.7.1 Calling Lambdas

Lambdas can be invoked by providing expressions to match it's parameter list.

```
Lambda l = (Int x) => return 2*x;

Int m = l(5); // m #=> 10

Lambda higher = (Int x, Int y) => {
    if(x>y){
        return x;
    } else{
        return y;
    }
}

Int larger = higher(10, 50); //larger #=> 50
```

2.7.2 Accepting Lambdas as Parameters

A Lambda may be passed to a function just as like other variable. Any lambda may be passed to a function that accepts a lambda.

```
Lambda double = (Int x) => return 2*x;

Int result = math(10, double); // result #=> 20

Lambda beginning = (String x) => return x.substring(0,2);

result = math(10, beginning);
 // this crashes because substring cannot be used on Int

public Int math(Int x, Lambda l){
    return l(x);
}
```

2.8 Built-in Libraries

This section contains the built in libraries and utilities for the language such as the array and list utilities and the system library

2.8.1 Collections

The Collection class extends the object class and is the base class for all data storage collections such as arrays and lists. The Collection class itself is abstract and therefore cannot be instantiated or used directly. It may, however, be sub-classed. The Collection class contains functions that must be implemented by sub-classes, ensuring that all collections have some base set of functions. The Collection class contains the following methods:

- public Int size()
- public void sort()
- public Bool contains(Object o)
- public Bool remove(Object o)
- public void concat(Object o)
- public Collection slice(Int start, Int end)
- public void slice!(Int start, Int end)
- public Int indexOf(Object o)
- public Object get(Int i)
- public Collection map(Lambda l)
- public void map!(Lambda l)
- public Collection select(Lambda l)
- public void select!(Lambda l)
- public Collection reduce(Object initialValue, Lambda l)
- public void reduce!(Object initialValue, Lambda l)
- public void each(Lambda l)

2.8.2 Arrays

An Array is a constant time, indexable collection of objects. An Arrays may be created for any type, including user created types. Arrays should be used when access time is important. Accessing an element in an array takes constant time. By default, Arrays are re-sizable. This behavior may be disabled with the setResizeable method. Arrays may be declared in the following ways:

```
Int[] arr = new Int[50];
// arr is an Array to hold integers with an initial size of 50.
String[] strArray = [1,2,4,5,42,7];
// strArray is an array holding Strings, with an initial size of 6
```

public Int size()

The .size() method returns the number of elements in the Array.

```
Char[] arr = ['A', 'G', 'h', 't'];
Int i = arr.size();
```

public void sort()

The .sort() method sorts the Array from lowest to highest value. Any object that has the `<`, `>`, and `==` operators defined may be sorted.

```
Int[] arr = [5,2,0,3,42,7];
arr.sort(); // arr #=> [0,2,3,5,7,42]

String[] s = ["abc", "zxy" "John"];
s.sort(); // s #=> ["abc", "John", "zxy"]
```

public Bool contains(Object o)

The .contains(Object o) method is used to check if an Array contains a specific value. This function may take up to $\mathcal{O}(n)$. If the array is known to be sorted then this operation takes $\mathcal{O}(log(n))$. The object must have the `==` operator defined on it.

```
Int[] arr = [0,1,5,7,42];
Bool b = arr.contains(2); // b #=> false
b = arr.contains(7); // b #=> true
```

public Bool remove(Object o)

The .remove(Object o) method is used to remove an object from an Array. Returns true if the object existed, false if it did not. The object must have the `==` operator defined on it.

```
String[] arr = ["hello","world","goodbye"];
Bool b = arr.remove("hello"); // b #=> true
b = arr.remove("Airplane"); // b #=> false
```

public void concat(Object o)

The .concat(Object o) method adds an object to the end of the Array.

```
Int[] arr = [1,4,7,2];
arr.concat(3); // arr #=> [1,4,7,2,3]
```

public Collection slice(Int start, Int end)

The .slice(Int start, Int end) method returns a new array containing the elements of the original array starting at index start and up to but not including index end.

```
Int[] arr = new [0,5,4,7,12];
Int[] arr2 = arr.slice(1,4); // arr2 #=> [5,4,7]
```

public void slice!(Int start, Int end)

The destructive .slice(Int start, Int end) method takes the elements in the Array from index start up to but not including index end, discarding the other elements.

```
Int[] arr = [2,6,12,9,5,7];
arr.slice!(2,5); // arr #=> [12,9,5,7]
```

public Int indexOf(Object o)

The .indexOf(Object o) method returns the index of Object o in the Array. If the object doesn't exist, -1 is returned.

```
String[] arr = ["hello", "goodbye", "world", "beaver"]
Int i = arr.indexOf("hello world"); // i #=> -1
i = arr.indexOf("world"); // i #=> 2
```

public Object get(Int i)

The .get(Int i) method returns the object at index i. The get method can also be accessed with the [ ] operator.

```
Char[] arr = ['a', 'b', 'd', 'e', 'f', 'g', 'h'];
Char c = arr.get(3); // c #=> e
c = arr.get(0); // c #=> a

Char c2 = arr[0];// c2 #=> a
```

public void set(Int i, Object value)

The .set(Int i, Object value) method sets the value of a particular index of an array. The set method can also be accessed with the [ ] operator.

```
Char[] arr = ['a', 'b', 'd', 'e', 'f'];
arr.set(0,'z');// arr #=> ['z', 'b', 'd', 'e', 'f']

arr[1] = 'x'; // arr #=> ['z', 'x', 'd', 'e', 'f']
```

public Collection map(Lambda l)

The .map(Lambda l) method applies a function to each element in the array and returns a new Array containing the new values, the original Array remains unaltered. Map accepts a lambda as a parameter. This lambda must accept one parameter and the return type must match the type of Array the result is being stored in. Map accepts either an in-line or declared lambda.

```
Int[] arr = [1,2,3,4,5];
Int[] arr2 = arr.map(i => return i*2); // arr2 #=> [2,4,6,8,10]
```

The map method does not necessarily have to return an Array with the same type as the initial Array.

```
Int[] arr = [1,2,3,4,5];
String[] arr2 = arr.map(i => {
   if(i%2 == 0){
      return "EVEN";
   } else{
      return "ODD";
   }
});
//at this point arr2 #=> ["ODD", "EVEN", "ODD", "EVEN", "ODD"]
```

public void map!(Lambda l)

The destructive .map(Lambda l) method applies a function to each element in the array. Map accepts a lambda as a parameter. This lambda must accept one parameter and the return type must match the type of the Array. Map accepts either an in-line or declared lambda.

```
String[] arr = ["hello","world","again", "porcupine"];
arr.map!(s => return s+=".");
 // arr #=> ["hello.","world.","again.", "porcupine."]
```

public Collection select(Lambda l)

The .select(Lambda l) method returns a subset of items from the Array where all items in the subset satisfy a particular condition. Select accepts a lambda as a parameter. This parameter must accept one parameter and return a Bool. Select accepts either an in-line or declared lambda.

```
String[] arr = ["hello", "world", "hat","happy"];
String[] result = arr.select(s => return s.contains('a'));
// at this point result contains ["hat","happy"]
```

public void select!(Lambda l)

The .select(Lambda l) method returns a subset of items from the Array where all items in the subset satisfy a particular condition, with the result being stored in the original Array. Select accepts a lambda as a parameter. This parameter must accept one parameter and return a Bool. Select accepts either an in-line or declared lambda.

```
Int[] arr = [1,2,4,5,7,10,14,21];
arr.select(x => return x%2 == 0);
//at this point arr contains [2,4,10,14]
```

public Object reduce(Object initialValue, Lambda l)

The .reduce(Object initialValue, Lambda l) method is used to reduce an Array to a single value. Reduce accepts a lambda. This lambda must accept two values: the accumulation value and the enumerated value. Reduce accepts an in-line or declared lambda.

```
Int[] arr = [1,2,4,5,7,10,13];
Int s = arr.reduce(0, (sum, x) => return sum+x); // s #=> 42
s = arr.reduce(100, (sum, x) => return sum+x); // s #=> 142
```

public void each(Lambda l)

The .each(Lambda l) method iterates through each item in the Array, and applies the lambda l to each element.

2.8.3 List

A List is a doubly linked list of objects. A List may be created for any type, including user created types. A List may be indexed, with an average retrieval time of $\mathcal{O}(n)$. Insertion at the head or tail takes $\mathcal{O}(1)$, as does deletion from the head or tail. A linked list has no set size limit. A List may be initialized in the following ways:

```
// note that no initial size is required for a List, unlike an
    Array
String{} lst = new String{};
Int{} i = {1, 2, 3, 5, 7, 19, 42};
```

public void push(Object o)

The .push(Object o) method adds an object to the end of a list. The push method can also be accessed with the += operator.

```
Int{} lst = {1,2,3,5,8};
lst.push(13);
//at this point lst #=> {1,2,3,5,8,13}
lst+=5;
//at this point lst #=> {1,2,3,5,8,13,5}
```

public Object pop()

The .pop() method removes the object at the end of the list and returns it.

```
Int{} lst = {1,2,3,5,8,13};
Int i = lst.pop(); // i #=> 13
//at this point lst #=> {1,2,3,4,5,7}
```

public void shift(Object o)

The .shift(Object o) method inserts an object at the beginning of the list.

```
Int{} lst = {1,2,3,4,5};
lst.shift(0);
//at this point lst #=> {0,1,2,3,4,5}
```

public Object unshift()

The .unshift() method removes the first item in the list and returns it.

```
Int{} lst = {1,2,3,4,5};
Int x = lst.unshift(); // x #=> 1
//at this point lst #=> {2,3,4,5}
```

public Int size()

The .size() method returns the number of elements in the List.

```
Char{} lst = {'A', 'G', 'h', 't'};
Int i = lst.size();
```

public void sort()

The .sort() method sorts the List from lowest to highest value. Any object that has the **<**, **>**, and **==** operators defined may be sorted.

```
Int{} lst = {5,2,0,3,42,7};
lst.sort(); // lst #=> {0,2,3,5,7,42}

String{} s = {}"abc", "zxy" "John"};
s.sort(); // s #=> {"abc", "John", "zxy"}
```

public Bool contains(Object o)

The .contains(Object o) method is used to check if an List contains a specific value. This function may take up to $\mathcal{O}(n)$.

```
Int{} lst = {0,1,5,7,42};
Bool b = lst.contains(2); // b #=> false
b = lst.contains(7); // b #=> true
```

public Bool remove(Object o)

The .remove(Object o) method is used to remove an object from an List. Returns true if the object existed, false if it did not.

```
String{} lst = {}"hello","world","goodbye"};
Bool b = lst.remove("hello"); // b #=> true
b = lst.remove("Airplane"); // b #=> false
```

public Collection slice(Int start, Int end)

The .slice(Int start, Int end) method returns a new List containing the elements of the original List starting at index start and up to but not including index end.

```
Int{} lst = new {0,5,4,7,12};
Int{} lst2 = lst.slice(1,4); // lst2 #=> {5,4,7}
```

public void slice!(Int start, Int end)

The destructive .slice(Int start, Int end) method takes the elements in the List from index start up to but not including index end, discarding the other elements.

```
Int{} lst = {2,6,12,9,5,7};
lst.slice!(2,5); // lst #=> {12,9,5,7}
```

public Int indexOf(Object o)

The .indexOf(Object o) method returns the index of Object o in the List. If the object doesn't exist, -1 is returned.

```
String{} lst = {"hello", "goodbye", "world", "beaver"};
Int i = lst.indexOf("hello world"); // i #=> -1
i = lst.indexOf("world"); // i #=> 2
```

public Object get(Int i)

The .get(Int i) method returns the object at index i. The get method can also be accessed with the [ ] operator.

```
Char{} lst = {'a', 'b', 'd', 'e', 'f', 'g', 'h'};
Char c = lst.get(3); // c #=> e
c = lst.get(0); // c #=> a
c = lst[2]; // c #=> d
```

public void set(Int i, Object value)

The .set(Int i, Object value) method sets the value of a particular index of a List. The set method can also be accessed with the [ ] operator.

```
Char{} arr = {'a', 'b', 'd', 'e', 'f'};
arr.set(0,'z');// arr #=> {'z', 'b', 'd', 'e', 'f'}

```

```
arr[1] = 'x'; // arr #=> {'z', 'x', 'd', 'e', 'f'}
```

public Collection map(Lambda l)

The .map(Lambda l) method applies a function to each element in the List and returns a new List containing the new values, the original List remains unaltered. Map accepts a lambda as a parameter. This lambda must accept one parameter and the return type must match the type of List the result is being stored in. Map accepts either an in-line or declared lambda.

```
Int{} lst = {1,2,3,4,5};
Int{} lst2 = lst.map(i => return i*2); // lst2 #=> {2,4,6,8,10}
```

The map function does not necessarily have to return an List with the same type as the initial List.

```
Int{} lst = {1,2,3,4,5};
String{} lst2 = lst.map(i => {
   if(i%2 == 0){
      return "EVEN";
   } else{
      return "ODD";
   }
});
//at this point lst2 #=> {"ODD", "EVEN", "ODD", "EVEN", "ODD"}
```

public void map!(Lambda l)

The destructive .map(Lambda l) method applies a function to each element in the List. Map accepts a lambda as a parameter. This lambda must accept one parameter and the return type must match the type of the List. Map accepts either an in-line or declared lambda.

```
String{} lst = {"hello","world","again", "porcupine"};
lst.map!(s => return s+=".");
// lst #=> {"hello.","world.","again.", "porcupine."}
```

public Collection select(Lambda l)

The .select(Lambda l) method returns a subset of items from the List where all items in the subset satisfy a particular condition. Select accepts a lambda as a parameter. This parameter must accept one parameter and return a Bool. Select accepts either an in-line or declared lambda.

```
String{} lst = {"hello", "world", "hat","happy"};
String{} result = lst.select(s => return s.contains('a'));
// at this point result contains {"hat","happy"}
```

public void select!(Lambda l)

The .select(Lambda l) method returns a subset of items from the List where all items in the subset satisfy a particular condition, with the result being stored in the original List. Select accepts a lambda as a parameter. This parameter must accept one parameter and return a Bool. Select accepts either an in-line or declared lambda.

```
Int{} lst = {1,2,4,5,7,10,14,21};
lst.select(x => return x%2 == 0);
//at this point lst contains {2,4,10,14}
```

public Object reduce(Object initialValue, Lambda l)

The .reduce(Object initialValue, Lambda l) method is used to reduce a List to a single value. Reduce accepts a lambda. This lambda must accept two values: the accumulation value and the enumerated value. Reduce accepts an in-line or declared lambda.

```
Int{} lst = {1,2,4,5,7,10,13};
Int s = lst.reduce(0, (sum x) => return sum+x); // s #=> 42
s = lst.reduce(100, (sum x) => return sum+x); // s #=> 142
```

public void each(Lambda l)

The each method iterates through each item in the List and applies the lambda l to each element.

2.8.4 Map

A Map is a key-value pair that may be indexed in constant time. All of the keys must be of the same type, and all of the values must be of the same type. A Map with Object keys means any object may be used as a key, similarly a map with object values means and object may be used as a value. A map may be initialized in the following ways:

```
Map<String, Int> m = new Map(); // a map with String keys and Int
    values
Map m2 = new Map(); // a map with Object keys and values
Map<String, Int> m3 = {"red" => 0, "blue" => 42};
```

public void set(Object key, Object value)

The .set(Object key, Object value) method is used to add a key-value pair to the map. If the key passed to set already exists in the map, it's value is overwritten with the new value. May also be accessed with the [ ] operator.

```
Map<Char, String> m = new Map();
m.set('A', "Apple");
m.set('B', "Butter");
//at this point m #=> {'A' => "Apple", 'B' => "Butter"}
m['T'] = "Toast"; //m #=> {'A' => "Apple,
                  //'B' => "Butter",
                  //'T' => "Toast"}
```

public Object get(Object key)

The .get(Object key) method returns the value associated with the given key. If no value is exists for that key, null is returned. May also be accessed with the [ ] operator.

```
Map<String, Bool> m = {"Arkansas" => true,
                       "Alabama" => false,
                       "Florida" => true};
Bool b = m.get("Alabama"); // b #=> false
b = m.get("Arkansas"); // b #=> true

b = m["Arkansas"];// b #=> true
```

public List keys()

The .keys() method returns a List containing all of the keys in the map.

```
Map<String, Bool> m = {"Arkansas" => true,
                       "Alabama" => false,
                       "Florida" => true};
String{} s = m.keys(); // s #=> {"Arkansas", "Alabama", "Florida"}
```

public List values()

The .values() method returns a List containing all of the unique values in the map.

```
Map<String, Bool> m = {"Arkansas" => true,
                       "Alabama" => false,
                       "Florida" => true};
Bool{} b = m.values();// b #=> {true, false}
```

2.8.5 System

The system library contains methods for interacting with the host operating system for things such as reading input, writing output, and changing system settings. Objects in the system object are accessible directly from the standard namespace.

public File StandardIn:

The StandardIn property contains the current standard in File, which by default is the command prompt. Any file can be assigned to this property.

public File StandardOut:

The StandardOut property contains the current standard out File, which by default is the command prompt. Any file can be assigned to this property.

public void resetStandardIn()

Resets standard in to the command prompt.

public void resetStandardOut()

Resets standard out to the command prompt.

Console

The Console object is used to read user input from as well as write output to standard in and out.

public Bool hasNext() The .hasNext() method returns true if standard in has more input, false if not.

public void write(String s) The .write(String s) method writes the string s to standard out. By default, standard out is the command prompt.

public void writeLine(String s) The .writeLine(String s) method writes the string s to standard out after appending a new line. This functions as a line break in the output. By default, standard out is the command prompt.

public String readLine() The .readLine() method reads standard in up to the next new line character and returns that string. By default, standard in is the command prompt.

public String readAll() The .readAll() method reads all input from standard in and returns that string. By default, standard in is the command prompt.

public String read() The .read() method reads input from standard in up to the next whitespace character and returns that string. By default, standard in is the command prompt.

public Int readInt() The .readInt() method reads input from standard in up to the next whitespace, parses it to an Int, and returns it. By default, standard in is the command prompt.

public Char readChar() The .readChar() method reads the next character from standard in and returns it. By default, standard in is the command prompt.

public String readTo(String s) The. readTo(String s) method reads input from standard in up to the next occurrence of s, and returns it. If s is not found, all input from standard in is returned. By default, standard in is the command prompt.

2.8.6 File

The file library is used to interact with files. If the file does not exist, it will be created. A Files contents are left intact. A file is initialized in the following way:

```
File in = new File("input.txt");
```

public Bool hasNext() The .hasNext() method returns true if you have not reached the end of the file, false if not.

public void write(String s)

The .write(String s) method writes the string s to the file.

public void writeLine(String s)

The .writeLine(String s) method writes the string s to the file after appending a new line. This functions as a line break in the output.

public String readLine()

The .readLine() method reads the file up to the next new line character and returns that string.

public String readAll()

The .readAll() method reads all input from the file and returns that string.

public String read()

The .read() method reads input from the file up to the next whitespace character and returns that string.

public Int readInt()

The .readInt() method reads input from the file up to the next whitespace, parses it to an Int, and returns it.

public Char readChar()

The .readChar() method reads the next character from the file and returns it.

public String readTo(String s)

The .readTo(String s) method reads input from the file up to the next occurrence of s, and returns it. If s is not found, all input from standard in is returned.

public void clear()

The .clear() method deletes the entire contents of the file.

2.8.7 Convert

The convert library is used to convert between one type and another.

toInt(String s)

The .toInt(String s) method accepts a String containing only a number and returns the integer value of that number.

```
String s = "42";
Int x = Convert.toInt(s); // x #=> 42
```

toInt(char s)

The .toInt(char s) method accepts a character that represents a a number and returns the integer value of that number.

```
Char c = '7';
Int x = Convert.toInt(c); // x #=> 7
```

toFloat(String s)

The .toFloat(String s) method accepts a String containing only a number and returns the floating point value of that number.

```
String s = "3.14159";
Float x = Convert.toInt(s); // x #=> 3.14159
```

toDouble(String s)

The .toDouble(String s) method accepts a String containing only a number and returns the floating point value of that number.

```
String s = "3.14159";
Double x = Convert.toInt(s); // x #=> 3.14159
```

Chapter 3

Code Examples

This section contains example code written in Emerald, as well as similar code written in Java and/or Ruby. The code is written using the conventions and idioms from the language it is written in.

3.1 Summing All Items in an Array

Emerald

```
Int[] arr = [1,2,3,4,5,6]

Int sum = arr.reduce(0, (sum, x) => return sum+=x);

\\at this point sum = 21
```

Java

```
Int[] arr = [1,2,3,4,5,6]
Int sum = 0;
for(Int i = 0; i < arr.length; i++){
   sum+=arr[i];
}

\\at this point sum = 21
```

3.2 Retrieving All of the Even Elements From an Array

Emerald

```
Int[] arr = [1,2,3,4,5,6,7,8,9];

Int[] evens = arr.select(x => return x%2 == 0);
//evens #=> [2,4,6,8]
```

Java

```
Int[] arr = [1,2,3,4,5,6,7,8,9];
List<Int> evens = new List<Int>();

for(Int i = 0; i < arr.length; i++){
   if(arr[i] % 2 == 0){
      evens.add(arr[i]);
   }
}
//evens #=> [2,4,6,8]
```

3.3 Find the Sum of All Digits in a List

The input string will come from standard in.

Emerald

```
Int[] input = Console.readLine().map(x => return Convert.toInt(x));
Int sum = input.reduce(0, (sum, x) => return sum+=x);
Console.writeLine(sum);
```

Java

```
Scanner in = new Scanner(System.in);
String[] input = in.nextLine().split("");
Int sum = 0;

for(Int i = 0; i < input.length; i++){
    sum+=Integer.parseInt(input[i]);
}
System.out.println(sum);
```

3.4 Find the Highest Number in a File

The File is named 'input.txt' and contains a comma separated list of positive numbers. The result of the program will be printed to standard out.

Emerald

```
Int[] input = new File("input.txt").readLine().map(x => return
    Convert.toInt(x));
Int highest = input.reduce(0,
            (highest, x) => return x > highest ? x : highest);
Console.writeLine(highest);
```

Java

```
Scanner in = new Scanner(new File("input.txt"));
String[] input = in.nextLine().split(",");

Int highest = 0;

for(Int i = 0; i < input.length; i++){
   if(Integer.parseInt(input[i]) > highest){
      highest = Integer.parseInt(input[i]);
   }
}
System.out.println(highest);
```

3.5 Summing Letters

Input will come from a file, 'input.txt', and each line will consist of a character and an integer value for that character, separated by a comma. The program will output the sum of 'a', 'r', 'k', 'a', 'n', 's', 'a', 's'.

Emerald

```
Map<String, Int> map = new Map();
File f = new File("input.txt");

String[][] arr = f.readAll().split("\n").map(x => return x.split
    (',');
arr.each(x => map[x[0]] = x[1]);

Int sum = "arkansas".reduce(0, (sum, x) => return sum+= map[x]);

Console.writeLine(sum);
```

Java

```
Scanner in = new Scanner(new File("input.txt"));
HashMap<String, Integer> map = new HashMap<>();

while(in.hasNextLine()){
   String line = in.nextLine();
   String key = line.substring(0,1);
   Int value = Integer.parseInt(line.substring(2));
   map.put(key, value);
}

String out = "arkansas";
Int sum = 0;
for(Int i = 0; i < out.length(); i++){
   sum+= map.get(out.substring(i, i+1));
}
System.out.println(sum);
```

3.6 CD Organizer

CD Organizer is a program that keeps track of a CD collection. It allows users to view their collection, add new CD's to it, remove CD's from it, and search for CD's.

3.6.1 Java

Main class

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

public class Main {
   public static String[] genres = { "Classical", "Rock", "Jazz", "
    Country", "Latin", "Pop", "Gospel",
   "Contemporary" };
   public static Scanner in = new Scanner(System.in);

   public static void main(String[] args) throws
    FileNotFoundException {

      Link head = new Link(null, null);

      readIn(head);

      showMenu();
      Int choice = in.nextInt();
      in.nextLine();

      boolean stale = false;

      while (choice != 9) {
         switch (choice) {
            case 1:
               enterNew(head);
               viewAll(head);
               stale = true;
               break;
            case 2:
               viewAll(head);
               break;
            case 3:
               search(head);
```

```
                break;
            case 4:
                delete(head);
                viewAll(head);
                stale = true;
                break;
            case 5:
                save(head);
                stale = false;
                break;
            default:
                System.out.println("Please enter a valid command.");
                break;
        }

        if (choice == 9) {
            System.out.println("Save your changes? (y/n):");
            if (in.nextLine().toUpperCase().equals("Y")) {
                save(head);
            }
        }

        showMenu();
        choice = in.nextInt();
        in.nextLine();
    }

}

private static void enterNew(Link head) {
    System.out.println("Enter the artist:");
    String artist = in.nextLine().toUpperCase();
    System.out.println("Enter the title:");
    String title = in.nextLine().toUpperCase();
    System.out.println("Enter the year:");
    Int year = in.nextInt();
    printGenres();
    Int genre = in.nextInt();
    in.nextLine();
    CD next = new CD(title, artist, year, genre);
```

```

      while (head.getNext() != null && (head.getNext().getCD().
   getArtist().compareTo(next.getArtist()) < 0 || (head.getNext().
   getCD().getArtist().compareTo(next.getArtist()) == 0 && head.
   getNext().getCD().getYear() > next.getYear()))) {
         head = head.getNext();
      }

      Link l = new Link(next, head.getNext());

      head.setNext(l);
   }

   private static void viewAll(Link head) {
      System.out.println("ARTIST TITLE GENRE YEAR");
      while (head.getNext() != null) {
         head = head.getNext();
         System.out.println(head.getCD());
      }
      System.out.println();// for extra line after output
   }

   private static void search(Link head) {
      System.out.println("Search by (1) Artist or (2) Genre?");
      Int choice = in.nextInt();
      in.nextLine();// eat new line after input

      if (choice == 1) {
         System.out.println("Enter Artist (all or partial name):");
         searchArtist(in.nextLine(), head);
      } else {
         System.out.println("Enter Genre:");
         searchGenre(in.nextInt(), head);
         in.nextLine();
      }
   }

   private static void searchGenre(Int genre, Link h) {
      while (h.getNext() != null) {
         h = h.getNext();
         if (h.getCD().getGenre() == genre) {
            System.out.println(h.getCD());
         }
      }
      System.out.println();
   }

   private static void searchArtist(String artist, Link h) {
      while (h.getNext() != null) {
         h = h.getNext();
         if (h.getCD().getArtist().contains(artist)) {
            System.out.println(h.getCD());
         }
      }
      System.out.println();
   }

   private static void delete(Link head) {
      System.out.println("Enter the title and artist of the CD to
   delete");
      System.out.println("Title: ");
      String title = in.nextLine().toUpperCase();
      System.out.println("Artist: ");
      String artist = in.nextLine().toUpperCase();

      while (head.getNext() != null) {
         if (head.getNext().getCD().getArtist().equals(artist) &&
   head.getNext().getCD().getTitle().equals(title)) {
            head.setNext(head.getNext().getNext());
         }
         head = head.getNext();
      }
   }

   private static void save(Link head) throws FileNotFoundException
    {
      PrintWriter p = new PrintWriter(new File("collection.txt"));

      while (head.getNext() != null) {
         head = head.getNext();
```

```
            p.println(head.getCD().toFile());
        }
        p.close();
    }

    private static void readIn(Link head) throws
     FileNotFoundException {
        Scanner f = new Scanner(new File("collection.txt"));

        while (f.hasNextLine()) {
            String[] input = f.nextLine().toUpperCase().split(",");

            CD n = new CD(input[0], input[1], Integer.parseInt(input
     [2]), Integer.parseInt(input[3]));

            head.setNext(new Link(n, null));

            head = head.getNext();
        }

    }

    private static void showMenu() {
        System.out.println("CD Organizer -- Enter your choice\n" + "
     1. Enter a New CD\n" + "2. View all CDs\n" + "3. Search for a CD
     \n" + "4. Delete a CD\n" + "5. Save\n" + "9. Exit the program\n"
     );
    }

    private static void printGenres() {
        System.out.println("Genre Number Genre Type");
        for (Int i = 0; i < genres.length; i++) {
            System.out.println(i + 1 + "             " + genres[i]);
        }
    }
}
```

CD class

```
public class CD {
   private String title, artist;
   private Int genre, year;

   public CD(String title, String artist, Int year, Int genre){
      this.title = title;
      this.artist = artist;
      this.year = year;
      this.genre = genre;
   }

   public String toString(){
      return title + " " + artist + " " + genre + " " + year;
   }

   public String toFile() {
      return title + "," + artist + "," + genre + "," + year;
   }

   public String getTitle() {
      return title;
   }
   public void setTitle(String title) {
      this.title = title;
   }
   public String getArtist() {
      return artist;
   }
   public void setArtist(String artist) {
      this.artist = artist;
   }
   public Int getGenre() {
      return genre;
   }
   public void setGenre(Int genre) {
      this.genre = genre;
   }
   public Int getYear() {
        return year;
    }
    public void setYear(Int year) {
        this.year = year;
    }

}
```

Link class

```
public class Link {
    private Link next;
    private CD cd;

    public Link(CD cd, Link next){
        this.cd = cd;
        this.next = next;
    }

    public Link getNext(){
        return next;
    }

    public CD getCD(){
        return cd;
    }

    public void setNext(Link l) {
        next = l;
    }
}
```

3.6.2 Emerald

Main program

```
String[] genres = { "Classical", "Rock", "Jazz", "Country", "Latin"
    , "Pop", "Gospel", "Contemporary" };

CD[] cds = new CD[0];
```

```

readIn();

Bool stale = false;

do{
   showMenu();
   Int choice = Console.readInt();
   Console.readLine();

   switch(choice){
      when 1:
         enterNew();
      when 2:
         viewAll();
      when 3:
         search();
      when 4:
         delete();
      when 5:
         save();
      default:
         Console.writeLine("Please enter a valid command.");
   }

} while(choice != 9);

public void enterNew(){
   Console.writeLine("Enter the artist:");
   String artist Console.readLine().toUpper();
   Console.writeLine("Enter the title:");
   String title = Console.readLine().toUpper();
   Consol.writeLine("Enter the year:");
   Int year = Console.readInt();

   Console.writeLine("Genre Number Genre Type");
   Int counter = 1;
   genres.each(x => Console.writeLine(counter++ + "              " +
   x));

```

```
    Int genre = Console.readInt();
    Console.readLine();
    cds+= new CD(title, artist, year, genre);

    cds.sort();
}

public void viewAll(){
    System.out.println("ARTIST TITLE GENRE YEAR");
    cds.each(x => Console.writeLine(x));
}

public void search(){
    Console.writeLine("Search by (1) Artist or (2) Genre?");
    if(Console.readInt() == 1){
        Console.readLine();
        Console.writeLine("Enter Artist (all or partial name):");
        String artist = Console.readLine().toUpper();
        cds.each(x => {
            if(x.Artist.contains(artist)){
                Console.writeLine(x);
            }
        });
    } else{
        Console.writeLine("Enter Genre:");
        Int genre = Console.readInt();
        Console.readLine();
        cds.each(x => {
            if(x.Genre == genre){
                Console.writeLine(x);
            }
        });
    }
}

public void delete(){
    Console.writeLine("Enter the title and artist of the CD to
    delete");
    Console.writeLine("Title: ");
    String title = Console.readLine().toUpper();
    Console.writeLine("Artist: ");
    String artist = Console.readLine().toUpper();

    cds.each(x => {
        if(x.Artist == artist && x.Title == title){
            cds.remove(x);
        }
    });
}

public void save(){
    File output = new File("collection.txt");
    output.clear();

    cds.each(x => output.writeLine(x.toFile()));
}

public void readIn(){
    File f = new File("collection.txt");

    f.readAll().split("\n").each(x => {
        String[] parts = x.split(',');
        cds+= new CD(parts[0], parts[1], Convert.toInt(parts[2]),
     Convert.toInt(parts[3]));
    });
}

public void showMenu(){
    Console.writeLine("CD Organizer -- Enter your choice\n" + "1.
     Enter a New CD\n" + "2. View all CDs\n" + "3. Search for a CD\n"
      + "4. Delete a CD\n" + "5. Save\n" + "9. Exit the program\n");
}
```

CD class

```
public class CD {
    private String title;
    property String Title{
        get{
            return title;
        }
        set{
            title = value;
        }
    }
    private String artist;
    property String Artist{
        get{
            return artist;
        }
        set{
            artist = value;
        }
    }
    private Int genre;
        property Int Genre{
        get{
            return genre;
        }
        set{
            genre = value;
        }
    }
    private Int year;
        property Int Year{
        get{
            return year;
        }
        set{
            year = value;
        }
    }

    public CD(String t, String a, Int y, Int g){
        title = t;
        artist = a;
        year = y;
        genre = g;
    }

    public String toString(){
        return title + " " + artist + " " + genre + " " + year;
    }

    public String toFile() {
        return title + "," + artist + "," + genre + "," + year;
    }

    public Bool < CD cd{
        return artist < cd.Artist && year < cd.Year;
    }

    public Bool > CD cd{
        return artist > cd.Artist && year > cd.Year;
    }

    public Bool == CD cd{
        return artist == cd.Artist && year == cd.Year;
    }
}
```

Chapter 4

Conclusion

After I completed the documentation for Emerald, I wrote the code for the examples section. This was something I had been looking forward to, and I was not disappointed. I found that the language was easy to write, made logical sense, and most of all, was extremely fun. I didn't tackle any huge problems, but I feel like I wrote enough code to say that there are no huge mistakes or problems in the language.

During the process of designing Emerald I had to make several difficult decisions regarding what to put in the language. One of the things I found most challenging was not making the language feel cluttered. I wanted the language to be flexible yet clean and uncluttered. Making it flexible was fairly easy: just have larger built-in libraries and multiple options to perform the same task, each with certain advantages. However, this quickly starts to feel cluttered. Making the language uncluttered was fairly easy as well: just reduce the size of the included libraries and make one way to do things. However, this can be limiting to the programmer. I worked very hard to try to find the right balance between these two extremes, although I feel as if I wasn't completely successful. There are some parts of the language such as the Collections, where I feel like there should be more options and more functionality in the built-in libraries. Overall, the language is clean and uncluttered, though I think it could have benefited from a bit more clutter in order to be even more flexible.

One of the other areas I ended up not being completely happy with is the operator overloading. Operator overloading is something that can be extremely powerful, and I wanted to ensure that it was accessible to programmers; however, I could never get it to feel right. Whenever I would use it, I always felt like something was wrong, or off. I believe part of that is because I haven't done much operator overloading in other languages, and so the whole concept is new, but I think part of it also came from an imperfect implementation of the concept.

The lambda section was one that I put off for as long as possible. I have seen too many terrible implementations and didn't want to end up with one in my language. In the end, the lambda section serves its purpose, but it's not perfect. To reduce the chance of creating something awful, I stripped the lambdas down to the bare minimum. I did not implement any checking, such as when passing a lambda to a function theres no way to know if you were passed a lambda matching what you're expecting. This, as well as many other common lambda features, I found extremely difficult to implement well, and opted to instead remove them from the language. I am not particularly happy with the way this section turned out, and if I were to do it again this would be one of the sections that I'd spend more time on.

4.1 Additions That Did Not Happen

There are several things that I wanted to be in the language that, for one reason or another, ended up not making the cut. Some were for technical reasons, some were simply from a lack of time.

Array slicing is something that I always found annoying in Java, yet exceptionally easy in Python and Ruby. Despite this, I ended up implementing a method-based version, similar to Java, rather than an operator based version, similar to Ruby. I do still feel like my implementation is better than the Java implementation, but not as nice as the Ruby implementation. I chose to do the method-based version because there was no precedent in Emerald for any operator-based slicing, and the Array would have been the only place it was used. So, instead of adding a new operator and a new set of rules to go with it, I decided it would be simpler to just use a method.

I originally wanted to add many more included utilities for the common types. I wanted to add things such as a math library for the number types, dictionary/word libraries for Strings, as well as a few other features. I ended up just not having enough time to implement these libraries.

Exceptions were a beast I did not even attempt to conquer. I had several ideas about how I wanted to handle exceptions, starting with how Java handles exceptions and making alterations to fix the many annoying aspects.

I realized very early on that, because exceptions are embedded into almost every part of a language, implementing an exception system could take just as long as developing the rest of the language, so I put them on the shelf and never was able to get back to them.