Shingle

Language Specification

Version 1.0.0

1. Introduction

Shingle is a simple, modern, object-oriented, and (strongly) type-safe programming language. Based on Java and C++, but differing in the following ways:

1. Brackets have been removed and instead `Start` and `End` determine when each function or class initiates and stops.
2. `Input` is used to get user data from keyboard rather than `System.in` or `cin`.
3. `Show` is used to output to the screen rather than `System.out.println` or `cout`.

1.1 Hello world

```
Start class HelloWorld
   Start static void Main()
      Show “Hello World”;
   End Main
End class HelloWorld
```

1.2 Program structure

The key organizational concepts in Shingle are as follows:

1. `Start` to determine when each function or class is initiated.
2. `End` to determine when each function or class is finished.
3. Indentations to help programmer see the structure more clearly.

This example

```
Start using namespace Acme.Collections
   Start public class Stack
      public Entry top;

      Start public void Push(object data)
         top = new Entry(top, data);
      End Push

      Start public object Pop()
         Start if (top = null)
            throw new InvalidOperationException();
         else
            object result = top.data;
            top = top.next;
            return result;
         End if
      End Pop


      Start private class Entry
         public Entry next;
         public object data;
```

```
            Start public Entry(Entry next, object data)
               this.next = next;
               this.data = data;
            End Entry
         End class Entry
      End class Stack
   End namespace Acme.Collections
```

declares a class named `Stack` in a namespace called `Acme.Collections`. The fully qualified name of this class is `Acme.Collections.Stack`. The class contains several members: a field named `top`, two methods named `Push` and `Pop`, and a nested class named `Entry`. The `Entry` class further contains three members: a field named `next`, a field named `data`, and a constructor.

1.3 Types and variables

There are two kinds of types in Shingle: ***value types*** and ***reference types***. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable.

1.4 Statements Differing from Java and C++

Statement	**Example**

Expression statement	`Start static void Main()`
	`   int i = 123;`
	`   Show i;`
	`   i++;`
	`   Show i;`
	`End Main`

`if` statement	`Start static void Main(string[] args)`
	`   Start if (args.Length == 0)`
	`      Show "No arguments";`
	`   else`
	`      Show "One or more arguments";`
	`   End if`
	`End Main`

`while` statement	`Start static void Main()`
	`   int x = 0;`
	`   Start while (x < 5)`
	`      Show x;`
	`      x++;`
	`   End while`
	`End Main`

`for` statement	`Start static void Main()`
	`   Start for (int i = 0;i < 5;i++)`
	`      Show x;`
	`   End for`
	`End Main`

1.5 Classes and objects

New classes are created using class declarations.

The following is a declaration of a simple class named `Point`:

```
Start public class Point
   public int x, y;

   Start public Point(int a, int b)
      x = a;
      y = b;
   End Point
End class Point
```

Instances of classes are created using the `new` operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance.

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

The memory occupied by an object is automatically reclaimed when the object is no longer in use. It is neither necessary nor possible to explicitly deallocate objects in Shingle, though you may be able to coax the garbage collector into action early.

1.5.1 Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are three possible forms of accessibility. These are summarized in the following table.

Accessibility	**Meaning**
`public`	Access not limited
`protected`	Access limited to this class or classes derived from this class
`private`	Access limited to this class

1.5.2 Fields

A field is a variable that is associated with a class or with an instance of a class.

```
Start public class Color
   public static Color Black = new Color(0, 0, 0);
   public static Color White = new Color(255, 255, 255);

   private byte r, g, b;

   Start public Color(byte x, byte y, byte z)
      r = x;
      g = y;
      b = z;
   End Color
End class Color
```

1.5.3 Methods

A ***method*** is a member that implements a computation or action that can be performed by an object or class.

The ***signature*** of a method must be unique in the class in which the method is declared.

1.5.3.1 Constructors

Shingle supports both instance and static constructors. An ***instance constructor*** is a member that implements the actions required to initialize an instance of a class. A ***static constructor*** is a member that implements the actions required to initialize a class itself when it is first loaded.

1.5.3.2 Properties

Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. Unlike fields, properties do not denote storage locations, but rather have ***accessors*** that specify the statements to be executed when their values are read or written.

1.5.3.3 Events

An ***event*** is a member that enables a class or object to provide notifications. Clients react to events through ***event handlers***. Event handlers are attached using the += operator and removed using the -= operator. The following example attaches an event handler to the `Changed` event of a `List<string>`.

```
Start public class Test
   static int changeCount = 0;

   Start static void ListChanged(object sender, EventArgs e)
         changeCount++;
   End ListChanged

   Start static void Main()
      List<string> names = new List<string>();
      names.Changed += new EventHandler(ListChanged);
      names.Add("Christine");
      names.Add("Luis");
      names.Add("Anthony");
      Show changeCount;    // Outputs "3"
   End Main
End class Test
```

1.6 Arrays

An ***array*** is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in an array, also called the ***elements*** of the array, are all of the same type, and this type is called the ***element type*** of the array. The following example allocates a one-dimensional array.

```
int[] a1 = new int[10];
```

The `a1` array contains 10 elements of type int.

2. Lexical structure

2.1 Programs

A Shingle ***program*** consists of one or more ***source files***. A source file is an ordered sequence of (probably Unicode) characters.

Conceptually speaking, a program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

2.2 Grammars

This specification presents the syntax of the Shingle programming language where it differs from Java and C++.

2.2.1 Lexical grammar where different from Java and C++

A Shingle program cannot allow a keyword or reserved word to be used as a user-defined identifier. It also does not allow for names to be re-initialized in the same scope.

2.2.2 Syntactic ("parse") grammar where different from Java and C++

A Shingle program contains a `Start` command at the beginning of each call, and an `End` command at the finish of each call. Initialization can only occur within these commands. Each `Start` command must have an `End` command in order to terminate. Identifiers of the same type can be concatenated or added together, but different types cannot.

2.2.3 Grammar notation

The lexical and syntactic grammars are presented using **BNF *grammar productions***. Each grammar production defines a non-terminal symbol and the possible expansions of that non-terminal symbol into sequences of non-terminal or terminal symbols. In grammar productions, *non-terminal* symbols are shown in italic type, and `terminal` symbols are shown in a fixed-width font.

The first line of a grammar production is the name of the non-terminal symbol being defined, followed by a colon. Each successive indented line contains a possible expansion of the non-terminal given as a sequence of non-terminal or terminal symbols. For example, the production:

while-statement:

`Start while (` *boolean-expression* `)`

embedded-statement

`End while`

defines a *while-statement* to consist of the token `Start`, followed by the token `while`, followed by the token "`(`", followed by a *boolean-expression*, followed by the token "`)`", followed by an *embedded-statement*, followed by the token combination `End while` to denote the termination of the *while-statement*.

When there is more than one possible expansion of a non-terminal symbol, the alternatives are listed on separate lines. For example, the production:

> *statement-list:*
>
> *statement*
>
> *statement-list statement*

defines a *statement-list* to either consist of a *statement* or consist of a *statement-list* followed by a *statement*. In other words, the definition is recursive and specifies that a statement list consists of one or more statements.

Alternatives are normally listed on separate lines, though in cases where there are many alternatives, the phrase "one of" may precede a list of expansions given on a single line. This is simply shorthand for listing each of the alternatives on a separate line. For example, the production:

> *real-type-suffix:* one of
>
> `F  f  D  d  M  m`

is shorthand for:

> *real-type-suffix:*
>
> `F`
>
> `f`
>
> `D`
>
> `d`
>
> `M`
>
> `m`

2.3 Lexical analysis

2.3.1 Line terminators

Line terminators divide the characters of a Shingle source file into lines.

> *new-line:*
>
> Carriage return character (`U+000D`)
>
> Line feed character (`U+000A`)
>
> Carriage return character (`U+000D`) followed by line feed character (`U+000A`)
>
> Next line character (`U+0085`)
>
> Line separator character (`U+2028`)
>
> Paragraph separator character (`U+2029`)

2.3.2 Comments

Two forms of comments are supported: single-line comments and delimited comments. ***Single-line comments*** start with the characters `//` and extend to the end of the source line. ***Delimited comments*** start with the characters `/*` and end with the characters `*/`. Delimited comments may span multiple lines, but do not nest.

2.3.3 White space

White space is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character.

> *whitespace:*
>
> Any character with Unicode class Zs
>
> Horizontal tab character (`U+0009`)
>
> Vertical tab character (`U+000B`)
>
> Form feed character (`U+000C`)

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens.

token:
- *identifier*
- *keyword*
- *integer-literal*
- *real-literal*
- *character-literal*
- *string-literal*
- *operator-or-punctuator*

All valid tokens in Shingle: `Start, End, Show, Input, int, double, char, string, boolean,` and any user-defined identifiers or names.

2.4.1 Keywords different from Java or C++

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the `@` character.

New keywords: one of

```
Start       End       Show      Input
```

Removed keywords:

```
do       then       System.out.println      cout     System.in      cin
```

3. Basic concepts

3.1 Application Startup

Application startup occurs when the execution environment calls a designated method, which is referred to as the application's entry point. This entry point method is always named `Main`, and can have one of either of the following signatures:

```
Start static void Main()...End Main
Start static void Main(string[] args)...End Main
```

As shown, program cannot begin or finish without the `Start` and `End` commands. Also shown is the ability to supply `Main` with an `array` of `string` arguments.

3.2 Application termination

Application termination returns control to the execution environment.

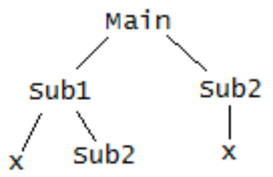
If the return type of the ***entry point*** method is `void`, reaching the outer-most `end` which terminates that method, or executing a `return` statement that has no expression, results in a ***termination status code*** of `0`. The purpose of this code is to allow communication of success or failure to the execution environment.

3.3 Scope

```
Start static void Main()
   int x = 5;
   Start void Sub1(int a)
      int x = a-1;
      Sub2(x);
   End Sub1
   Start Sub2(int a)
      return a-2;
   End Sub2
   Sub1(x);
End Main
```

Static scoping: Reference to x is to `Main`'s x
Dynamic scoping: Reference to x is to `Sub1`'s x

`Symbol Table Diagram:`



3.4 Automatic memory management

Shingle employs automatic memory management, which frees developers from manually allocating and freeing the memory occupied by objects. Automatic memory management policies are implemented by a ***garbage collector***. It differs from Java and C++ in the following ways:

1. Within a program, garbage collection and deletion of identifiers to free up space can be forced.
2. Within a given scope, identifiers are kept until they are no longer needed or already used.
3. Non-used identifiers at termination of program are kept until program is re-initialized.

4. Types

Shingle types are divided into two main categories: ***Value types*** and ***reference types***.

4.1 Value types (different from Java and C++)

```
int, double, char, string, boolean
```

4.2 Reference types (differing from Java and C++)

```
&int, &string
```

5. Variables

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. Shingle is a type-safe language

5.1 Variable categories

`int`: any number without a decimal point can be stored

```
   Start static void Main()
      int a = 0;
      int b = -5000;
      int c = 5000;
   End Main
```

`double`: any number with a decimal point can be stored

```
   Start static void Main()
      double a = .5;
      double b = -5000.12345;
      double c = 5000.12345;
   End Main
```

`char`: any single character can be stored

```
   Start static void Main()
      char a = ‘A’;
      char b = ‘\t’;
      char c = ‘#’;
   End Main
```

`string`: any combination of characters can be stored, or even one character

```
   Start static void Main()
      string a = “Hello World”;
      string b = “Hi, this is fun…\t NOT!”;
      string c = “\n”;
   End Main
```

`boolean`: true or false can be stored

```
   Start static void Main()
      boolean flag = true;
      boolean otherflag = false;
   End Main
```

6. Paramater Passing

6.1 Method

Methods that strictly use In parameters are looping functions such as `for` and `while`. Since the internal number is only used as a looping organizational tool, there is no return of the value.

Methods that strictly use Out parameters are any methods that return a value.

Methods that use InOut parameters are methods that take an inputted value, change it, and eventually output a value of the same type. These methods also tend to use In and Out parameters.

Pass by value is used to initiate former parameters. These involve calls to subroutines that give a value back up to the method that called it.

Pass by reference is used by InOut parameters, but sends the address of the value to the called subprograms.

6.2 Examples

Pass-by-Value example:

```
 Start public void Main()
   int a = 5;
   Start public Sub3(int x)
      return x-3;
   End Sub3
   Sub3(a);
End Main
```

Activation record:

Main -	Local variable: a = 5
Sub3 -	Local variable: x
	Parameter: int
	Return address

Pass-by-reference example:

```
Start public void Main()
   int a = 5;
   Start public Sub(int x, int &y)
      return x-y;
   End Sub
   Sub(a,3);
End Main
```

Activation record:

Main -	Local variable: a = 5
Sub -	Local variables: x, &y
	Parameter: int
	Return address

7. Conversions

A ***conversion*** enables an expression to be treated as being of a particular type.

7.1 Implicit conversions (are bad in Shingle)

```
Start static void Main()
   boolean flag = true;
   Show “True is actually int = 1.\n”;
   Show “False is actually int = 0.”;
End Main
```

7.2 Explicit conversions (much better in Shingle)

```
Start static void Main()
   Point p1 = new Point(10,10);
   Show “I have explicitly told p1 what it was going to be.”
End Main
```

Further explicit conversions can be done using a number of either pre-defined and user-defined classes by use of constructors in those classes.

8. Statements

Regarding statements, Shingle differs from Java and C++ in the following areas:

1. There are no brackets.
2. `Start` command must be immediately before.
3. `End` command must be immediately after.

9. Example Programs

Example program 1:
```
Start class Shape
   private int height, width;
   private Color color;
   Start Shape(int ht, int wid, Color clr)
      height = ht;
      weight = wid;
      color = clr;
   End Shape
   Start void setSize(int ht, int wid)
      height = ht;
      width = wid;
   End setSize
   Start string getSize()
      return height + “ “ + width;
   End getSize
End class Shape
```

Example program 2:
```
Start static void Main()
   string[] ary1 = new string[10];
   Start for (int i = 0;i < ary1.Length;i++)
      ary1[i] = i.toString();
      Show ary1[i];
   End for
End Main
```

Example program 3:
```
Start static void Main()
   string str = “Hello”;
   char chr;
   Start for (int i = 0;i < str.Length;i++)
      chr = str.charAt(i);
      Show chr;
   End for
End Main
```

Example program 4:
```
Start static void Main(string[] args)
   Show “Please input an integer.”;
   Input int x;
   Show “Your number:” + x;
   x++;
   Show “Your number + 1:” + x;
End Main
```

10. Conclusion

I believe that my programming language, Shingle, is an improvement over Java and C++ because it is easier to read but still carries the same implementation and functionality found in Java and C++. It is most closely related to Java due to my familiarity with Java, and only a few things from C++, which is similar to Java anyway, at least in functionality.

So, as far as improving it, I first took out the brackets in order to make it easier to just keep typing, rather than stopping all the time to put right and left brackets in their appropriate places. I always got so frustrated with that aspect, especially in the beginning, because it would take extra time just to put them in. So, instead of brackets, I added a `Start` command to the beginning of everything, as well as an `End` command, because it makes it easier to read. I just think it makes it more organized to have clearly defined commands such as `Start` and `End`, that almost anyone could understand and any good typist could type with ease, instead of brackets.

I also changed `System.out.println` and `cout` to `Show`, because the Java version is a very long command to type just to get output, and the C++ version isn't very clear. When using Java, every time I wanted to output something to the screen, I had to type `System.out.println`, and it just got very annoying to have to type so much to do such a simple task. When I came across other languages that just use print or write, it just made sense to me. There's no need for type-checking, because all it's doing is throwing it on the screen to show it to whoever wants to see it. That's why I wanted to use something like that, and came up with the `Show` command, which is much shorter to type and makes sense.

Also because it was unclear, I changed `cin` to `Input`, and got rid of `System.in` due to its high use on the `Scanner` class just to get input from the user. Granted `cin` is shorter to type than `Input`, but when I first started learning C++, it was very unclear what that meant. And `System.in` has to rely on a context to hold it, such as `Scanner`, so I don't feel that makes it any easier. You'd have to make an instance of `Scanner`, and then use `System.in` within it, so I felt like taking the middle ground and allowing for a quick access method without having to import other classes, and that's where the method `Input` comes in.

Therefore, I believe Shingle takes the usefulness of Java and C++ and adds more of a readable structure to it to make it a definite improvement. It takes all the good and gets rid of the bad, especially of Java. Now it is easy to read, easy to write, easy to understand, and easy to implement. The `Start` and `End` methods provide a very understandable structure, the `Input` method is very easy to utilize, and the `Show` method just makes sense. All these things combined, I feel that Shingle comes out on top above Java and C++.