Pineapple Game IDE 1.0

Functional Specification for PineDL

Last Updated: July 22. 2009

Overview

PineDL 1.0 is the programming language for game development used in the Pineapple Game IDE 1.0.

PineDL 1.0 is case-sensitive.

Target Audience

All Pineapple users will need PineDL to develop games. Therefore, the target audience for PineDL is the same as the target audience for Pineapple.

Script Example

An example of a PineDL is the following:

```
class Example{
     public this(){
          //This is a comment
     }
}
```

In the example above, we can see a class named “Example” declared. It has a public constructor in it and a comment.

Dynamic Typing, Static Typing and Constants

PineDL allows both dynamic and static typing.

Static typing means that a variable, once declared with a type, can not ever change it.

In dynamic typing, however, a given variable may have any type at any time, regardless of its original declaration.

Unless a type is specified in the variable declaration, PineDL assumes a variable is dynamically typed.

```
var x = 3; //x is dynamically typed
x = "C"; //Valid

int y = 3; //y is statically typed
y = "C"; //Invalid

z = 3; //Invalid. z was not declared at all.

const a = 1; //a is constant
a = 2; //Invalid. Value of “a” can't be changed.
```

Classes

PineDL support object-oriented programming. Classes can be defined and single-inheritance is supported with the “extends” keyword.

```
class ClassName [extends BaseClass]{
     //Class Implementation
}
```

```
class Shape{
}

class Square extends Shape{
}

class Circle extends Shape{
}
```

Simple Constructor And Field Example

Consdering the case of a “Player” that must have “Cards”. A possible PineDL implementation would be:

```
class Player{
     public Card[] cards;

     public this(){
          cards = new Card[];
          foreach(int i = 0; i < 10; i++)
               cards[i] = new Card();
     }
}
class Card{
     //Card Data
     //Since no constructor is provided, a public constructor is
used automatically
}
```

Constructors are defined as:

```
[Access Modifier] this( [arguments] ){ /*code*/ }
```

If no access modifier keyword is provided, “public” is used.

Method Example

The type of the method's return value can be defined both statically or dynamically, just like variables, and the same applies to the type of each argument.

```
class Player{
     public function doSomething(bool a, b){
          //a is statically typed, while b is dynamically typed.
```

```
          if(a) return 3;
          return new Player();
     }
     public function yetAnotherMethod(a) : int{
          //The argument a can have any type.
          return 3; //The statement "return new Player();" would
throw a compile error here.
     }
}
```

Methods that have no type specified for return values are declared as:

```
[Access Modifier] function FunctionName([Arguments]){
//Implementation
}
```

Methods with a type specified for return values are declared as:

```
[Access Modifier] ReturnType FunctionName([Arguments]){
//Implementation
}
```

Methods with no return value are declared as:

```
[Access Modifier] void FunctionName([Arguments]){
//Implementation
}
```

If no access modifier is provided, then "public" is used.

Argument Lists

Methods, functions and constructors have argument lists.
Argument lists can be empty or can be defined as:

```
FirstArgument [,SecondArgument [, ThirdArgument [...]]]
```

Each dynamically typed argument is defined as:

```
[var] argumentName [= value]
```

Each constant argument is defined as:

```
const [type] argumentName [= value]
```

If no type is specified, any typed can be passed.
Each statically typed argument is defined as:

```
type argumentName [=value]
```

If no default value is specified, the null constant is used.

Constants

There are a few types of constants:

- Null constant. The keyword “null”.
- Boolean constants. The keywords “true” and “false”.
- Integer constants.
- Float constants(in formats like 1.2, .3 or 4e3)
- Character constants(in the 'C' format)
- String constants(in the “Hello \”World\”!” or @”C:\Users\Username\Documents\” formats)

Types

The following types are allowed:

- Any class name
- A valid type, followed by the '[' and the ']' characters(such as int[][])
- “void”(for function return values only)
- “string”, “int”, “char”, “bool” and “float”

```
class X{}

function test(){
X a; //Valid
int b; //Valid
void c;//INVALID. Void is only accepted as function return value
int d = 2; //Valid
int e = null; //INVALID.
string f = null; //Valid
int[] g = null; //Valid. Although integer itself is not
nullable, an array of integers is.
string[][] h; //Valid

var h = 'a'; //Valid. Dynamic typing
h = g; //Valid. Dynamic typing
}
```

Common Operations

PineDL has several operations.

```
Function test(){
var x;
x = 3+3; //6
x = 3-2; //1
x = -x; //-1
x = 3*3; //9
x = x/2; //4, since x is currently an integer
x = 3%2; //1
x = true && false; //false
x = true || false; //true
x = 2 & 4; //2
x = 2 | 4; //6
```

```
x = 2 ^ 4; //6
x++; //6. x becomes 7
++x; //8. x becomes 8
x = !false; //true
x = 3 is int; //true
x = 3 > 2; //true
x = 3 < 2; //false
x = 3 >= 3; //true
x = 3 <= 4; //true
x = 3 == 4; //false
x = 3 != 4; //true
x = 4 << 2; //16
x = 4 >> 2; //1
x = 1==1?2:3; //2
}
```

Operations like “x += 2;” are also available.

Conditional Operations And Loops

PineDL also supports the “if/else” system as well as “while” and “for”.

```
function test(){
if(1==1){
     //This gets executed
}
else{
     //This does not get executed
}
for(int i = 0; i < 10; i++){
     //This gets executed ten times.
}
var j = 0;
while(j < 10){
     //This gets executed 10 times.
     j++;
}

k = false;
j = 0;
while(true){
     k = !k;
     if(k){
          continue; //Checks the condition again and goes back
to the beginning of the loop
     }
     if(j==5){
          break; // Breaks the loop.
     }
     j++;
}
}
```

Exception Handling

PineDL also supports exceptions, including “try/catch” statements and “throw” statements as well, but not “finally” statements.

```
function test(){
     try{
          try{
               throw new Exception();
          }
          catch(e){ //Dynamic typing.
          }
     }
     catch(Exception j){} //Static typing
}
```

Only classes that inherit from Exception can be thrown.

Constants

While dynamic variables are declared as:

```
var name [ = expression ];
```

and static variables as:

```
type name [ = expression ];
```

Constants are declared as:

```
const name = expression;
```

Once declared, neither the type nor the value of a constant may change. In constants, the type is never explicitly specified.

This And Super

When within a class, one can use the keywors “this” and “super”. “this” can be used to access the current class instance. The “super” keyword allows calling methods from the base class(very useful for overidden methods).

One may not use “this” nor “super” in a function declared outside of a class nor in a static context.

Also, “super(arguments)” is an optional first statement of PineDL class constructors which specifies a constructor of the base class to call first(by default, the “super( )” constructor is used).

Static

While fields and methods may be declared with the “static” keyword are associated with a class type rather than a specific class instance.

Variable fields, variable constants and methods can all be static, but none of these are static by default.

Functions, Methods And Access Modifiers

While methods are declared within classes and must have an access modifier keyword associated, while other functions are declared outside classes and have no access modifier keyword.

There are three access modifier keywords in PineDL:

- public
- private
- protected

Their meaning closely matches other programming languages such as Java, C++ and C#.

Abstract Methods

Abstract methods are declared just like regular methods, but instead of /*implementation*/, they just have “;”.

```
class A{
     function AbstractMethod();
}
```

Any class that contains abstract methods is considered abstract.

Any class that inherits from an abstract class and does not implement the abstract methods is abstract. All others are not abstract.

```
class A{ function a(); }
class B extends A{ function b(); }
class C extends B{ function b(){} }
```

In the example above, C is abstract because although it implements “b”, it does not implement “a”.

One can not use “new Type( )” if “Type” is an abstract class. Note, however, that Type may have a constructor.

PineDL Scripts

A PineDL script may contain one or more classes and one or more non-class functions.

```
class A{}
class B extends A{}
function c(){}
```

Multiple Inheritance

PineDL supports limited multiple inheritance.

```
class A extends B, C{}
```

For the statement above to be valid PineDL, B and C can't have any fields, constants or non-abstract methods with the same name.

Therefore, if both B and C inherit from D, the statement above is only valid if D has only abstract or static methods(no non-static constants nor non-static fields nor non-abstract non-static methods).

Note that, for initialization purposes, the constructor for B is called before the constructor for C when initializing A.

If B and C have any fields, constants or non-abstract methods with the same name, then B and C are considered “incompatible”.

```
class A{
     function a();
     function b();
     function c(){}
}
class B{
     function a(){}
     function c();
}
class C extends A, B{
     //Valid, because A and B are not incompatible.
     //Although both have “a” and “c” methods, A's a() and B's
c() are abstract, therefore they do not conflict with each
other.
}
```