JAWS *Language Reference Manual*

1. Introduction

The JAWS programming language was designed to provide a platform for casual and experienced game designers to rapidly prototype and develop space shooter genre video games. The main focus behind the language was to eliminate the necessity for mastery of complex graphics and game physics implementation to allow the game designer to solely focus on game play design. Thus, the syntax and structure of the JAWS language strives for simplicity and ease of use.

2. Lexical Conventions

2.1 Comments

Comments are segments of code that the programmer wants the compiler to ignore. Comments in the JAWS programming language are limited to one per line. Comments must begin with "//" and everything else on that line up to the carriage return "/n" is treated as a comment.

2.2 Whitespace

In addition to comments, blanks, tabs, and carriage returns are ignored except where they are used to separate tokens. Tokens must be separated by at least one of the above.

2.3 Identifiers

Identifiers consist of a sequence of letters and digits and must start with a letter. In addition to letters, the underscore "_" can also be used and counts as an alphabetic. Identifiers represent the names that are given to declared data types that are used. It is suggested that identifiers begin with a lowercase letter. However, this is not enforced syntax.

2.4 Keywords

The following list of identifiers are reserved in the JAWS language and may not be used otherwise.

int	entity
string	movement
boolean	attack
double	point
constant	map
if	show
else	hide
for	true
to	false
by	
while	

2.5 Constants

JAWS allows the use of 3 types of constants: int, string, and double. Constants are defined by typing the "constant" prefix before an identifier. For example, to define a constant called "HIGH_SCORE":

```
constant string HIGH_SCORE = 100;
```

Constants retain their initial values and may not be changed during the execution of the program. It is suggested that constants be named with all uppercase letters to distinguish them from identifiers. However, this is not enforced syntax.

2.6 Separators
The characters used as separators in JAWS are:
( ) { } , ;

2.7 Tokens
Tokens are separated into identifiers, constants, keywords, operators, and other separators. Tokens are defined greedily meaning the token is the longest un-terminated recognized token type. All tokens must be separated by whitespace.

3. Data Types
There are nine types of data in JAWS: int, double, string, boolean, point, entity, movement, attack, and map.

3.1.1 int
Ints will be 32-bit two's compliant integers.

3.1.2 double
Doubles will be 64-bit double precision floating point numbers.

3.1.3 string
Strings will be a character string beginning with a " and ending with a ". JAWS will not support the single char data type so for single characters strings can be used in the same way.

3.1.4 boolean
Booleans will be a true or false value.

3.1.5 point
Since locations and points are so prevalent in JAWS, the point type is standard. Points consist of two ints, an x and a y, which uniquely define the point. The distance operator may be applied to point types, facilitating the calculation of distance between points and entities. Point type attributes may be directly accessed using the 's operator.

Points are declared like a normal type and its x and y are defaulted to 0. Point has a built-in initializer that takes in an int x and an int y which may be called to initialized x and y. Here is an example:

```
// point p will be at x = 0, y = 0
point p;

// point p will be at x = 6, y = 4
point p = point(6,4);
```

Point Type Attributes:

Type	Name	Description
int	x	x-coordinate of the point
int	y	y-coordinate of the point

3.1.6 entity

Entities are objects that interact with each other during game play that are displayed to the user. Each entity has a set of attributes that describe fully the entity's behavior and look. Entities can be players, monsters, powerups, labels, and anything else that could appear on the map. It is up to the programmer to correctly define their behavior.

Entity types in JAWS are all implemented as arrays. This facilitates the easy duplication of a multiplicity of entities without repeated declaration and initialization. When an entity class is created, all entities in its array are created with the same values. Each entity in the object array may be referenced and accessed individually thereafter. When an entity is referenced with an index specified, only that specific entity in the array is changed. When an index is left out, all entities in the array are affected.

Entity classes have special built-in fucntions which facilitate game programming. The ~ (distance) operator can be applied on two entity classes and returns the distance between those two entities. Entities also have show and hide functions. Show is called to display the entity and to load its initializer. Hide is called to unload the entity from display.

All attributes in a created entity class are set to default but can be initialized to different values in its initializer. To define the entity's initializer, define a function with the name of the entity without a return type.

The syntax for declaring an entity class and its initializer is as follows:

```
// entity declaration
entity name[multiplicity];

// entity initializer
name()
{
      [attribute] = newvalue;          // initializing attributes
      .
      .
      [type] name = value;       // adding user-defined variables
      .
      .
}
```

An example of creating a monster type called small_alien of multiplicity 5, and setting its constructor to lay them across horizontally, and creating user defined variables called hitpoints and pointValue:

```
entity small_alien[5];

small_alien()
{
      // individually set each small_alien's location
      for i = 0 to 5 by 1
      {
            small_alien[i]'s location = point(i*2, 0);
      }

      // set all small_alien's hitpoints to 10
      int hitpoints = 10;
```

```
      int pointValue = 100;
      show();
}
```

Entity Type Attributes:

Type	Name	Description
int	height	the height (in cells) of the entity on the map
int	width	the width (in cells) of the entity on the map
point	location	the location of the entity
string	image	address of a supported image file to be used
attack	weapon	the attack class that the entity is equipped with
movement	ai	the movement class to used to describe this entity
string	label	a string to display using the entity as a label

Entity Type Functions:

Name	Description
~	calculates the distance between two entities
show()	displays the entity on screen and executes the entity's initializer, the entity's movement ai also starts execution
hide()	unloads the entity from the screen and all execution associated to that entity is halted

3.1.7 movement

The movement type basically describes how an entity moves about the screen and how it determines its moves either randomly or based upon certain conditions. Movement types are defined individually and may take in parameters. Within the movement type is the description of the movement.

Movement types exist as attributes in entities. The location of the entity can be referred to by the movement type directly. Any other locations that the movement type wants to interact with must be passed in as parameters.

The movement type is a loop which keeps running for the specified entity. That means that defined movements are always looped and executed until that entity is unloaded with the hide command.

The syntax for declaring a movement class and its body is as follows:

```
movement name(type1 param1, ... , typeN paramN)
{
      // movement body
}
```

An example of a movement called retreat which makes this entity move north when the passed in entity is within 10 units of it. Otherwise, the entity keeps moving south.

```
movement charge(point oppLocation)
{
      if ( (location ~ oppLocation) < 10)
      {
          location's incy(1);
      }
      else
```

```
	{
		location's decy(1);
	}

}
```

JAWS implements a built-in movement type called keyboard, which moves an entity based on keyboard input. It will be of type movement and be called keyboard.

3.1.8 attack

Attack types represent an entity's offensive arsenal. It describes both how that entity's attack looks like, and also when and how it attacks.

Attack Type Attributes:

Type	Name	Description
string	image	the location of the image to be used
int	width	width in cells of the attack
int	height	height in cells of the attack

An attack type also contains three functions, two of which must be defined by the user, the shooting() and collision() functions. The last is the initializer which can be left blank. Shooting() describes the rate, speed, and direction of the attack. Collision() defines what action to take when this attack collides with the passed in entity.

```
attack name
{
	// initializer
	name() {
	[attribute] = newvalue;
	.
	.
	.
	}
	shooting() {

	}
	collision(entity source) {

	}
}
```

3.1.9 map

Map types represent individual levels. Each map can be specified with a width and height, and also a background image.

Map Type Attributes:

Type	Name	Description
int	width	width of the map
int	height	height of the map
string	background	image to be used for the map

Map Type Functions:

Name	Description
show	loads the map onto the display screen
hide	unloads the map from the display screen

Map types also have initializers with the same name. Map types are declared and initialized as follows:

```
map level1;

// initializer
level1()
{
      width = num;
      height = num;
      background = "image1.jpg";
}
```

3.2 Type Casting and Interaction

3.2.1 int and double

JAWS automatically type casts between int and double in assignments. An int may be defined using a double and vice versa. When declaring an int with a double, the result is always rounded down, meaning the precision is thrown away. For example, int a = 3.9 will leave a with the value of 3.

Int and double interaction during other operators is described below in the operators section.

3.2.2 string, int, and double

JAWS automatically type casts between strings and ints and doubles in assignments. Strings declared with an int or double will be accepted and converted to the corresponding character string. Int and doubles that are assigned strings will also automatically convert given that the string is in valid int or double form.

```
int i = "4";          // valid
double d = "5.2";     // valid
string s = 3;         // valid
string s = 3.2;       // valid

int i = "4.0";        // invalid
double d = "5.a";     // invalid
```

String interaction with other operators is described below in the operators section.

3.2.3 point, entity, movement, ai, map

Point, entity, movement, ai, and map types may not be used interchangeably and cannot be cast from one type to another. They all represent distinct data and no similarities exist enough between them to warrant type casting from one type to another. These types may be passed into functions where their individual attributes may be modified.

Interaction of these types with other operators is described below in the operators section.

4. Expressions

4.1 Expressions
General expressions refer to a combination of identifiers, keywords and operators that usually equate to a data type.

4.1.1 Mathematical Expressions
Mathematical expressions are combinations of identifiers and operators that take on the value of either an int or a double after evaluation. For example:

```
4.3         // valid mathematical expression
5 + 7 * 6   // valid mathematical expression

int a = 5; // invalid mathematical expression
```

4.1.2 Boolean Expressions
Boolean expressions are combinations of identifiers and operators that evaluate to a true or false value.

4.2 Statements
Statements are a combination of primary expressions, operators, and keywords usually ending in a semicolon. Statements usually constitute a line of code in JAWS.

5. Operators
The supported operators in jaws are assignment, additive, multiplicative, boolean, relational, equality, concatenation, distance, and class accessor.

5.1.1 Assignment Operators
Assignment operators assign an identifier to a new value. Some automatic type casting occurs between int, double, and strings. No type casting occurs for the other data types. Type casting is described in depth in the previous section.

The supported assignment operators are:
```
=      *=
+=     /=
-=
```

5.1.2 Additive/Multiplicative Operators
Additive and multiplicative operators may only be applied to int, double, and mathematical expressions. The + operator when applied to strings is a concatenation operator, not additive or multiplicative. If the operators are applied to all ints, the result will be an int. If the operators are applied to all doubles, the result will be a double. If the operators are applied to both ints and doubles, the result will be a double. For example:

The supported additive and multiplicative operators are:
```
+      *
-      /
%
```

5.1.3 Relational Operators

Relational operators may only be applied to int, double, and mathematical expressions. When relational operators are applied to both double and int, the comparisons will be done using the double value of the int.

The supported relational operators are:

```
>       >=
<       <=
```

5.1.4 Boolean/Negation Operators

Boolean and negation operators may only be applied to booleans and boolean expressions. Boolean operators return a boolean.

The supported relational operators are:

```
&&      !
||
```

5.1.5 Equality Operators

Equality operators may only be applied to data of the same types. Equality operators return a boolean. Equality applied to int, double, string, boolean, and point will be evaluated according to data. Equality in entity, movement, ai, attack, and map types are not evaluated based on their attributes but according to address.

The supported equality operators are:

```
==
!=
```

5.1.6 String Concatenation Operator

The string concatenation operator + can only be applied to strings. No automatic casting is done when using this operator.

```
"abc" + "def"          // valid
"abc" + 5              // invalid
```

5.1.7 Entity Distance Operator

The entity distance operator ~ can only be applied to the point and entity data types. The return value is a double with the absolute distance between the two arguments.

5.1.8 Type Accessor

The 's type accessor is used to access attributes and functions within the point, entity, movement, attack and map classes and thus can only be applied to those types.

```
class's attribute
class's function( )
```

5.2 Operator Precedence

The order in which operators are applied is described below, with operators at the top evaluated first, and operators on the same line evaluated according to their left to right order in the expression. Note that parentheses ( ) may be applied to give precedence to their contents in expressions. Precedence is listed from highest priority to lowest priority

Priority	Operator	Function
1	`'s`	type accessor
	`()`	function call
	`[]`	entity array index
2	`!`	negation
3	`~`	distance
4	`* / %`	multiplication, division, modulus
5	`+ -`	addition, subtraction
	`+`	string concatenation
6	`> >=`	greater, greater/equal
	`< <=`	less, less/equal
7	`==`	equal
	`!=`	not equal
8	`&&`	logical and
9	`||`	logical or
10	`=`	assignment
	`+= -=`	
	`*= /=`	

6. Declarations

6.1 User defined variables

Primitive data types int, double, string, boolean are declared…

Without initialization:

```
type name;
```

With initialization:

```
type name = initial-value;
```

Uninitialized int default to 0, double to 0.0, stings to “”, boolean to true.

6.2 User defined functions

User defined functions are declared as follows:

```
return-type function-name(type1 param1, … , typeN paramN)
{
      //function body
      return retValue;
}
```

The function must include a return statement returning an expression of the specified return type.

7. Conditionals and Loops

7.1 Coniditonal If…Else

Conditional if statements must have their then and else blocks separated by brackets { }. They should follow the form:

```
if boolean-expression
{
      //code block
}
else
{
      //code block
}
```

7.2 While Loops
While loops check the specified condition before the execution of the block and keep repeating as long as the condition is true. While loops follow the form:

```
while boolean-expression
{
      //code block
}
```

7.3 For Loops
For loops take an int, an end int, and an optional increment

```
for int to endInt by increment
{
      //code block
}
```

8. Coordinate System

The coordinate system in JAWS will center the point with x = 0 and y = 0 to the bottom left of the screen and increasing towards the top and right. This will model normal geometric coordinate systems and provide ease of use for the programmer.

9. Program Structure

The structure of a JAWS program consists of two main parts, the declarations and the engine. All programs in JAWS must be written in this context and all code should fall under either the declarations or engine sub parts.

```
// My JAWS Program
declarations
{
      // declarations
}

engine
{
      // engine code
```

```
}
```

9.1 Declarations
All constants, variables, entity, movement, attack, point, map types must be declared in the declarations segment of the program. The order in which they are declared is ignored so they can be declared in any order.

9.2 Engine Code
The engine segment contains all code relating to how the game is run during play. It is subdivided into two parts, the main loop and user-defined functions. The main loop is the game loop which keeps executing during gameplay. Here is how it is broken down:

```
engine
{
      main()
      {
            //main game play code
      }

      // user-defined functions
      // .
      // .
      // .
      // etc
}
```

9.1 A Sample JAWS Program
Here we try to illustrate the basic look and feel of a JAWS program with a sample of a complete and functional JAWS program. This program will simulate a game which

```
// Sample JAWS Program
// By Justin Lu
// "Attack of the Aliens" v1.0

declarations
{
      // lets create the map for our game
      map mainLevel;

      mainLevel()
      {
            // it will be a 50 x 50 map and
            //use a background called "bg.jpg"
            width = 50;
            height = 50;
            background = "bg.jpg";

      }

      // lets create an entity for our controllable player
```

```
entity player[1];
player()
{
      // our player will be 2x2 cells in size
      height = 2;
      width = 2;

      // lets make him start at the lower center of our map
      location = point(width/2 , 0);

      // we will equip him with a basic gun attack class,
      // to be defined later
      weapon = gun;

      // we will use a built-in default movement
      // called keyboard to allow input
      ai = keyboard;

}

// now lets create some enemy aliens, lets make 5
entity alien[5];

alien()
{
      // they will be 2x2 in size
      width = 2;
      height = 2;

      // lets lay them out across the top of the map
      for int i = 0 to 5 by 1
      {
            alien[i]'s location = make point( mainLevel's width/5 *
            i , mainLevel's height);
      }

      // we will set their AI to dumb_ai,
      // a movement class we will define
      ai = dumb_ai;
}

// we will now create the attack class gun that the player uses
attack gun
{
      gun()
      {
            image = "cool_laser.jpg";
            width = 1;
            height = 1;
      }
      shooting()
      {
```

```
                // we will use a builtin function
                // which uses keyboard input
                // when enter is pressed, our attack will go upwards
                if keyboardInput() == "/n"
                {
                      y++;
                }
          }
          collision(entity source)
          {
                // when our attack collides
                // with enemies of type alien,
                // lets unload them
                if source == alien
                {
                      source's hide();
                }
          }
     }

     // lets set our enemy AI now
     // it will be a dumb AI which just moves the enemy downwards
     movement dumb_ai( )
     {
          location's y -= 1;
     }
}

engine
{
     main()
     {
     // our game will have no engine code, lets just let it run
     // simple enough game, that all actions can be defined in
declarations
     // if we wanted to extend the game play,
     // we would need to add engine code
     }
}
```