MotorScript
Specification

Sem Nijenhuis
Stephen Nedd

Compilers & Operating Systems

MotorScript

Contents

Saxion University of Applied Sciences
Stephen Nedd 484629
Sem Nijenhuis 470066

MotorScript Language Description

MotorScript is a new programming language based on the well-known **Java** language. MotorScript was created because the developers felt that Java could use a *performance* boost.

FEATURES

Classes, Arrays

MotorScript **does** not have support for classes but they are called **Chassis**.
Arrays will also be an extra feature included in **MotorScript**.

Static Typing

MotorScript is statically typed to make the language more readable, meaning that variables that have already been declared as a particular type **cannot** later be changed to another type. Trying to do this will result in an error being thrown and the program stopping.
It is statically typed to lower the number of program crashes caused by type errors and make the language more user-friendly.
An added benefit of static typing is faster than dynamically typed languages due to not checking types while executing a program.

Type Inference

Types are inferred in MotorScript, meaning you do not need to specify what type of variable something is. The language will do it for you.

Strong typing

The MotorScript language uses **Strong Typing**, meaning the types of all variables are checked at compile time.

Functions

Users will be able to define functions in MotorScript. ( ) A function can have any number of parameters and will be able to return a value.

Operator Overloading

MotorScript will **not** support operator overloading

Saxion University of Applied Sciences
Stephen Nedd 484629
Sem Nijenhuis 470066

MotorScript

Expressions and Statements

In MotorScript, assignments are expressions.

Simple assignment statement:

Inline4 engine is 4#

Syntactic Sugar

MotorScript will support some syntactic sugar similar to the Java language.
MotorScript supports the following:

	Java	MotorScript
Increment	++	++
Decrement	--	--
Compound Assignment	+=	overtake

Examples Programs

HelloWorld:

Java:

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}

Output:
        Hello, World!
```

MotorScript:

```
frame HelloWorld {
    open lockup empty leading(String[] args) {
        System.out.burnout(“Motor, World!”)#
    }
}

Output:

        Motor, World!
```

Saxion University of Applied Sciences
Stephen Nedd 484629
Sem Nijenhuis 470066

While Loop:

Java:

```
void bla() {
    while (true) {
        System.out.println("bla");
    }
}

output:
Endless loop of “bla”
```

MotorScript:

```
void bla() {
    while (true) {
        burnout("bla")#
    }
}

output:
Endless loop of “bla”
```

Saxion University of Applied Sciences
Stephen Nedd 484629
Sem Nijenhuis 470066

Method Calls:

Java:

```
public class MyClass {
   public void myMethod() {
         System.out.println(“myMethod, at your service!”);
   }

   public static void main(String[] args) {
         myMethod();
   }
}

Output:
         myMethod, at your service!
```

MotorScript:

```
open frame MyClass {
    open empty myMethod() {
         System.out.burnout(“MotorScript can call methods too!”)#
    }

    open lockup empty leading(String[] args) {
         myMethod()#
    }
}

Output:
         MotorScript can call methods too!
```

Lanuage MotorScript

Arithmetic Operators

Java Operators	MotorScript Operators	Operand Type	Description
+	upshift	Arithmetic	+ (Addition)
-	downshift	Arithmetic	- (Subtraction)
*	nitro	Arithmetic	* (Multiplication)
/	divide	Arithmetic	/ (Division)

Assignment Operator

Java Operators	MotorScript Operators	Operand Type	Description
=	is	Assignment	Assigning

Saxion University of Applied Sciences
Stephen Nedd 484629
Sem Nijenhuis 470066

Relational Operator - Comparison Operator

Java Operators	MotorScript Operators	Operand Type	Description
==	==	Relational Operators	Is Equal
!=	!=	Relational Operators	Not Equal

Logical Operator

Java Operators	MotorScript Operators	Operand Type	Description
&&	AND	Logical Operator	Returns true if, both statements are true
||	OR	Logical Operator	Returns true if one of the statement is true

Looping & Condition

Java Loops & Checking	MotorScript Loops & Condition	Java Loop or Condition statement	Description
While Loop	While(<condition>) { }	While() { }	

Operator # - end sentences

Java Operators	MotorScript Operators	Operand Type	Description
;	#		End sentence

MotorScript

Primitive Data Types

Java Operators	MotorScript Operators	Operand Type	Description
Int (4 bytes)	Inline4	Primitive Data Type	Stores whole numbers from -2,147,483,648 to 2,147,483,647
Float (4 bytes)	Flat4	Primitive Data Type	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
Boolean (1 bit)	EV	Primitive Data Type	Stores true or false values

Non-Primitive Data Types

Java Operators	MotorScript Operators	Operand Type	Description
Class	Frame	Non-Primitive Data Type	Frame Vehicle{ }
Object	Body	Non-Primitive Data Type	Vehicle car = new Vehicle()
String	Sentence	Non-Primitive Data Type	Sentence abc = "Educative is an excellent tutor";
Array	Series	Non-Primitive Data Type	Series a1 = {“a”, “b”}

Modifiers

Java Operators	MotorScript Operators	Operand Type	Description
Public	Open	Modifiers	Public class <t>{}
Private	Closed	Modifiers	Private class <t>{}
Static	Lockup	Modifiers	Public static void String(args){}
Void	Empty	Modifiers	Void class{}
Extends	TowBar	Modifiers	Class extends
Implements	Trailer	Modifiers	Class implements interface

Special Feature - Array

Creating an Array

1. Initialize which type your going to use and use open & closed blocks
    - "inline4[]"
        - Only integers are allowed.
2. Give it an Identifier (name)
    - "inline4[] cookies"
3. Initialize the arraysizetype with a type and a number.
    - "inline4[] cookies is new"
        - The type has to be the same as first initialize type.
        - The number can't be 0 or a negative number.
4. Put within blocks after the type your array size and end the statement with a #
    - "inline4[] cookies is new inline4[10]#"

Result: inline4[] cookies is new inline4[10]#

Search and Adjust element in Array

By using the identifier from the array and index number your able to print or adjust an element from the array.

Search element

1. First start with the identifier
    - "array"
2. Inside blocks put the index number from the array you want to use.
    - "array[0]"
3. Now you could print this out by using the burnout statement or use it as new integer.
    - burnout(array[0])#
    - inline4 answer is array[0]#

Result: inline4[] array is new inline4[10]# array[0] is 7# burnout(array[0])#

-

Adjust element

1. First start with the identifier
    - "array"
2. Inside blocks put the index number where you want adjust a number.
    - "array[0]"
3. Use the assignment operator (is) and add the new number you want to have on that location and end the statement.
    - "array[0] is 7#"

Result: array[0] is 7#

Add element to a Array

1. First start with the identifier
    - "array"
2. Inside blocks put the index number where you want to add a number.
    - "array[3]"
3. Use the assignment operator (is) and add the number you want to add.
    - "array[3] is 22#"

Result: array[3] is 22#

MotorScript

Scanner

With the scanner we can use the input of the user.

Create and use the scanner

To creat

1. Create the scanner with an identifier
    - Scanner input#
2. Now you can use the scanner by calling the identifier.scan
    - input.scan
        - only String & Integers are allowed

Result: answer is input.scan#

Whileloop

The user is able to use while loop to run certain code until specific condition is met.

```
while(answer1 != guess) {
   burnout(feedback)#
   answer1 is input.scan#
}
```

Create and use a while loop

1. Start to the declare the while statement and put inside parentheses your condition
    - While (1>2)
        - The result of the statement must be a Boolean
2. Put something inside brackets what you want to run during the condition
    - while(1>2) {burnout(123)#}
        - The word break is not made, the user need to end their own loop.

```
Result: while(1>2) {
   burnout(123)#
}
```

If statement

The user is allowed to use if statements in motorscript to set specific rules

Create and use if statement

2. Start to the declare the if statement and put inside parentheses your condition
    - If(example)
        - The result of the statement must be a boolean
3. Now place within blockstatement(brackets) your code you want to run
    - if(example) {burnout("You made a mistake")#}
4. It is possible to add else if statement but this is not required
    - if(example) {burnout("You made a mistake")#} else { burnout(test)#}

Result: if(example) {

```
burnout("You made a mistake")#
} else {
burnout(test)#
}
```

Print statement

By using burnout the user can print something out.

Create and use print statement

1. First declare burnout and put within parentheses the code you want to have printed
    - Burnout(1)#
        - The input cant be null

Functions

In MotorScript we have support for functions like in Java. However, In our language you can only create functions without arguments and that return void.

Declare Functions

1. Enter the function keyword (funky), Use the return type void (currently only one supported) and an Identifier followed by parentheses arguments, then opening brackets.
    - funky void *func* () { *< function block >* }

Call Functions

You can call a function after It has been declared with the Identifier and parentheses followed by an end symbol. The function block Is then executed.

- *func* ()#

Saxion University of Applied Sciences
Stephen Nedd 484629
Sem Nijenhuis 470066