# Table of Contents

Table of Contents  
Overview  
Types  
  Primitive Types  
    int  
    float  
    bool  
  Reference Types  
Lexical Conventions  
  Comments  
  White Space  
  Reserved Words  
  Identifiers  
  Literals  
  Operators  
Expressions  
  Evaluation and Result Computation  
  Expression Type  
  Evaluation order  
  Lexical Literals  
  Arithmetic Operations  
  Relational Operations  
  Method invocation  
  Array Access  
  Assignment  
Classes  
  class declaration  
  Constructor Declaration  
  Field Declaration  
  Method Declaration
Statements 15
  Expression Statement 15
  Declaration Statement 16
  Control Flow Statements 16
    If else 16
    Looping: for 17
    Looping: foreach 17
    Looping: while 17
  Branching: break, continue, and return 18
    Break 18
    Continue 18
    return 18
  Method 19
  Empty Statement 19
Overview

Espresso is a hybrid object oriented language that incorporates some functional language features, such as supporting foreach statement blocks. The language is inspired by Java. The key goal of the project is to practice design of a simple language and use Ocaml to implement it. Like Java, Espresso is at its best when coding feels intuitive and allows programmers to worry about high-level programming rather than semantics - that's our problem. The basic operators, conditional statements, looping and datatypes will be supported with object oriented behaviour, along with some new lambda related features. Espresso is a general use language. Coders can use it to practice object oriented programming as well as create simple programs instinctively. The code of Espresso possesses similarities in terms of structure with Java, and will be compiled to LLVM. For purposes of simplification, Espresso will not support access or scope modifiers such as public, private, protected and default.
Types

Primitive Types

int

Integer is a type that stores a supplied value in 4 bytes. Intuitively, the type should be used to store whole decimal number values. Integers can store decimal values ranging from $-2^{31}$ to $2^{31}-1$. Initialization is shown below.

```c
int latte = 4;
int mocha = 2147483647;
int cappuccino = -9;
```

float

Float is a type that stores a supplied value in 4 bytes. Intuitively, the type should be used to store all real numbers to which the Integer type is insufficient. In practical terms, this will mean fractional numbers and numbers of greater magnitude than is supported by Integers. Initialization is shown below.

```c
float latte = 4.5;
int mocha = -9.0;
```

bool

The Boolean type is a binary indicator that can be either True or False. Booleans can also be null. Additionally, a boolean can be compared to or initialized as an Integer that is assigned as 0 or 1, or assigned as a 1 or 0.

```c
bool latte = true;
bool coffee = false;
bool mug = (coffee == 0);  // mug is True
```

char
The Character type is a single alphabetic ASCII character between single quotes. The range is ‘a’ - ‘z’, ‘A’ - ‘Z’. Initialization is shown below.

```cpp
char roast = 'c';
```

void

The void type is used as a placeholder to imply that a method will not return a value. All methods have to have a return type, so void must be used if there is no concrete return desired. Below is a sample method declaration:

```cpp
void methodName (<formals_opt>) {
    //method body
    //reading comments again??
}
```

Reference Types

Arrays

Arrays are datatypes that store into memory 0 or more items of a specific type in an indexed manner. Sample array declarations are shown below.

```cpp
int fantasticArr[10];        //An array of ten Integers
bool true_false_arr[1];      //An array of one Boolean value
```

Arrays can be initialized at the time of declaration. The values in the array must be supplied in brackets and separated by commas. A example of this is shown below.

```cpp
float fractionArr[3] = {1.1, 4.5, -2.2};
```

A user can also store values at a specific index later on.
fractionArr[2] = 0.07; //The array is now {1.1, 4.5, 0.07}

Strings

A string is a class. An instance of a string object contains an array of the primitive datatype char. Espresso supports three methods to manipulate strings: substring(), charAt(), length().

```java
String sen = "Fresh cup of coffee.";
String short_sen = sen.substring(3,6); //short_sen holds "shc"

//note that substring() is inclusive on both parameters
char myChar = sen.charAt(0); //char holds 'F'
int numLetters = sen.length(); //numLetters holds 20
```
Lexical Conventions

Comments
Supported comments are of two types:

- Single Line Comments -

///Start typing here

- Multiple Line Comments -

/*
None of this matters.
Nothing really matters.
This code, like you, is an insignificant speck.
*/

White Space
White spaces in Espresso are comprised of single space characters, tab spaces, page breaks and line ending characters. These are ignored by the Espresso compiler (we'll call it brew) with the primary purpose being that of acting as a separator for tokens. That is, one space serves the same purpose as several lines of space.

Reserved Words
Keywords are reserved and cannot be used as regular identifier names.
Identifiers

An identifier is a sequence of letters, digits, and underscores. It can only begin with a lowercase letter. Identifiers are essentially the names of variables, methods, and classes. They are case-sensitive.

Literals

Literals are syntactic depictions of the values of integers, characters, booleans or strings. They indicate the actual representation of values with the program context.

- Boolean literals - Two possible boolean literals:
  - true
  - false

- Integer Literals

  These are of the primitive type int. They are numeric values that do not comprise any decimal component.

- Floating Point Literals

  These are expressed as decimal fractions and consist of types like 0.56, 1.23, etc.

- Character Literals

  Character literals are contained between a pair of single quotes.
• String Literals
  String literals start with “", followed by any number of characters and end
  with “”. No newline character can occur with the string unless correctly
  escaped.

• Null literals
  Null literal refers to a single value that implies a particular does not refer to
  any object or value: `null`.

Operators
Operators will include relational, boolean and logical operators, described in greater
detail in the expressions section.
Expressions

A large component of the work in Espresso is done in the form of evaluation of expressions. An example of this is the evaluation of variable assignments of the following type:

```java
int a = 10;
```

Evaluation and Result Computation

When an expression is evaluated, the eventual result will be one of the following:

- A variable
- A value or a component of a larger expression
- Void - in the instance of void functions, for example.

The single case in which an expression can be nothing (or void) is in the case of its utilization as a return type for a particular method/function that has no return type as it does not return any value on completion of execution.

On the other hand, if the expression evaluates to a variable (which falls under the subcategory of an identifier), then within the overall evaluation of the expression, the value of the variable is applied.

As such, for both values and variables, expressions evaluate to values, that in themselves may be the final result or may be a component of a larger expression, depending on how they are nested.
Expression Type

Expressions are often used in the form of assignment operations of the following type:

\[
x = a + 1;
\]

Hence, the evaluation of expressions in Espresso is such that the result is of the same type as the variable it is assigned to. Often the range of operations that can be incorporated in specific expressions are only possible given that they are semantically valid for the types of values/variables that are present in the expression.

Eg.

```java
int a;
a = 5 % "e";
```

The above code snippet makes no sense in the context of Espresso, as the `%` operator holds no meaning in relation to the String "e", even though ordinarily it would indeed have valid context in relation to `a` and `5`.

Evaluation order

The evaluation order that is internally employed by Espresso is universally left-to-right.

Lexical Literals

Lexical literals indicate the actual representation of fixed, unchanging values that are the smallest unbroken unit that can be evaluated as one, and as such can not be further diluted into expressions.

The mapping between the various literals and their corresponding literals are as follows:

- Integer literals map to **int** datatype.
- Floating literals map to **float** datatype.
- Boolean literals map to `boolean` datatype.
- Character literals map to `char` datatype.
- String literals map to the `String` datatype.
- Null literal has no datatype; rather it is a single applicable value: `null`.

**Evaluation** of a literal is essentially a direct mapping.

### Arithmetic Operations

Arithmetic operators include the following:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Adds values on both sides</td>
</tr>
<tr>
<td>-</td>
<td>Subtracts value on the right side from the value on the left</td>
</tr>
<tr>
<td>*</td>
<td>Multiples values on either side</td>
</tr>
<tr>
<td>/</td>
<td>Divides left hand operand by the right hand operand</td>
</tr>
<tr>
<td>%</td>
<td>Divides left hand operand by the right hand operand and computes remainder</td>
</tr>
<tr>
<td>**</td>
<td>Computes left operand raised to the value of the right operand</td>
</tr>
</tbody>
</table>

These are only possible when both operands are primitive types like `int` or `float`. They are all binary operands and follow left to right associativity.

### Relational Operations

The value on evaluation of any relational expression always results to boolean. Equality comparisons can only be between similar types.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Returns true if both operands equal</td>
</tr>
<tr>
<td>!=</td>
<td>Subtracts value on the right side from the value on the left</td>
</tr>
<tr>
<td>&gt;</td>
<td>Returns true if left operand greater, else false</td>
</tr>
<tr>
<td>&lt;</td>
<td>Returns true if right operand greater, else false</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Returns true if left operand greater than or equal to right, else false</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Returns true if left operand lesser than or equal to right, else false</td>
</tr>
</tbody>
</table>

**Method invocation**

If the invoked method has a return type void, then void(no result) is returned and no return type is expected. Else, the value with the datatype specified in the method signature, is returned. A return statement is expected at the end of the invoked method that has to return a value.

**Array Access**

Array access is done in the form of an array reference (a variable or identifier) followed by an index (or position) enclosed in square brackets. The index must be of type int, if not, Espresso will automatically redirect it to the closest integer value less than the specified index (flooring to avoid possibility of hitting an index out of bounds greater than array size).

**Assignment**

The assignment operator is of the type ‘=‘. This is the only type of assignment supported in Espresso.

**Classes**

**class declaration**

A class declaration defines how it is implemented, it has fields and methods.

Class Declarations. A class can extends some other class or none
Class A{
}
Class SubA extends A{
    // fields
    // methods
}

Constructor Declaration
Constructor declaration is a specific method which has the same name of class and does not have return type.

Field Declaration
Field declaration is an expression.

Method Declaration

Class BankCount{
    int saving;
    String name;
    BankCount(String n,int a){
        name = n;
        Saving = a;
    }
    boolean withdraw(int a){
        if( a < 0 )
            return false;
        else if(saving > a ){
            saving -=a;
            return true;
        }
        else{
            return false;
        }
    }
    boolean deposit(int a ){
        if (a < 0 ){
            return false;
        }
        saving +=a;
        return true;
    }
}
Statements

Expression Statement
An expression statement contains an expression, and ends with a semicolon.

expression;
Declaration Statement

Declaration Statement can declare basic type array or class;

```java
int a;
float b;
String sentence;
boolean flag;
String [] sentences;
int [] datas;
Class A;
```

Control Flow Statements

If else

If else flow control can have else or not.

```java
if (expr1) expr2;

if(expr1){
    expr2;
}else{
    expr3;
}
```

Following is two examples.

```java
bool gt(int a, int b ){
    if (a >= b);
        return true;
    return false;
}

Int max(int a, int b){
    if (a >= b)
        return a;
    else
        return b;
}
```
Looping: for

Looping for works as following:

```java
for(expr1; expr2; expr3){
    expr4;
}
```

An example to print number from 0 to 9.

```java
for(int i = 0; i < 10; i++){
    print(i);
}
```

Looping: foreach

Looping foreach works as following:

```java
foreach (typ item : array) {expr1;}
```

An example to print an array.

```java
Int [] data = {1,2,3,4,5};
foreach(int item: data){
    print(item);
}
```

Looping: while

Looping while works as following:

```java
while(expr1){
    Expr2;
}
```

An example to print 0 to 9
```java
int i = 0;
while( i < 10){
    print(i);
}
```

Branching: break, continue, and return

Break

Break with break the closest looping. An example only print the connected positive number for each array. The output should 1, 2, 5, 6, each for one line.

```java
int i = 0;
int [][] m = {{1, 2, -3, -4}, {5, 6, -7, -8}}
while ( i < 2){
    foreach (int item from m[i]){  
        if (item > 0)
            print item;
        else
            Break;
    }
}
```

Continue

Continue will pass current expression of looping. An example only print all positive number for an array.

```java
int [] data = {1, 2, -3, 4};
for (int i = 0; i < 4; i++){
    if(data[i] < 0)
        continue;
    print(data[i]);
}
```

return

Return will return the function, it can return nothing or an expression.

```java
return;
return expression;
```

An example of return the first positive number of an array.
```java
int firstPositiveNumber(int[] data) {
    for (int i = 0; i < 4; i++) {
        if (data[i] > 0)
            return data[i];
    }
    return -1;
}
```

**Method**

Method statement works as following:

```java
returntype functionname( typ a, typ b){
    exprs;
    return returntype;
}
```

Return type can be void or basic type or an array or class;

```java
Class A{}
Class B1 extends A{}
Class B2 extends A{}

A factoryMethod(String t){
    if (t=="B1")
        return new B1;
    if (t=="B2")
        return new B2;
    return A;
}
```

**Empty Statement**

Empty Statement is nothing but just semicolon.