Chapter 3

Language Reference Manual

3.1 Introduction

This manual describes the *Espresso!* language. *Espresso!* programs must be written in Unicode on a UNIX platform.

3.2 Program Structure

An *Espresso!* program consists of zero or more 'cards':

```
card myCard {
	statements;
	user-defined functions;
}
```

A card can be defined as a window or layout of an applet. Each card will consist of a group of statements and/or user-defined functions.

Cards are an abstraction to help the programmer visualize the concept of an applet. Cards may be nested. Only one card may be displayed in a window at a time, and are stored as a stack. Statements and user-defined functions are described in the proceding sections.

3.3 Lexical Conventions

3.3.1 Tokens

There are classes of tokens: identifiers, keywords, string literals, operators, and other separators. Spaces, tabs, newlines and comments separate tokens but are otherwise ignored.

3.3.2 Comments

The characters /* indicate the start of a comment, which terminates with the characters */. Nesting of comments is not supported, nor is commenting within other tokens.

3.3.3 Identifiers

An identifier is a sequence of letters and digits. The first character must be a letter. Any successive character may be either a letter (uppercase or lowercase), digit (0..9), or the underscore '_' character.

Identifiers are case sensitive, for instance `"ABC"` is different from `"abc".` An identifier can be arbitrarily long. *Espresso!* places no explicit limit on identifier length, however they cannot exceed the available memory of the machine where the program is being developed.

3.3.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
card              while             do              if              elseif
else              break             number          string          print
method            return            true            false           paint
setBackground     setDimension      repaint         TEXT            TYPE
JUSTIFICATION     LENGTH            WIDTH           COLOR           POLYVNUM
AFOCIX            AFOCIY            BFOCIX          BFOCIY          FGCOLOR
BGCOLOR           SIZE              ACTION          SOURCE          UNIQUE
```

3.3.5 Numbers

A number consists of a string of digits with an optional sign `'-'` and an optional decimal point '.'. All built-in mathematical operations performed on numbers assume base-10 format.

3.3.6 Strings

A string is a sequence of characters enclosed by double quotes: "*string"*. A double quote inside the string is indicated by two consecutive double quotes. The second double quote is ignored in the final token.

3.3.7 Other tokens

- Spatial delimiters:

( ) { } [ ] ; : , .

- Mathematical operators:

+ - * / %

- Boolean operators:

> < ! == >= <= !=

- Assignment operators:

= += -= *= /= %=

3.3.8 Types

This language has minimal types to ensure simplicity and ease of use while resembling Java type syntax. Memory does not need to be explicitly allocated for typed variables and they can be created at anytime throughout the program. Implemented types are as follows:

- number : 64 bit IEEE floating point format
- string : arbitrarily long array of Unicode characters
- object : see table 3.2.1 for object types and descriptions
- method : user-defined function that must have a return value and may have zero or more parameters. The scope of the function is limited to the card in which it is defined. See section 3.6 for more details and default parameter values.

Object	**Description**
TextBoxes	An array of 1 or more boxes where strings are displayed and/or modified.
Buttons	An array of 1 or more buttons with titles and built in ActionListener capabilities.
ChkBoxes	An array of 1 or more check boxes with built in ActionListener capabilities.
Images	An array of 1 or more images that can be imported into the applet from .jpeg or .gif files.
Shapes	An array of 1 or more geometric shapes. Polygons are defined by their vertices and ellipses are defined by their radii and foci.
List	A static list of strings that can be displayed in a pulldown menu within the applet.
ScrlBar	A scroll bar displayed on the right and/or bottom edge of the applet.
Menus	A pulldown menu from the title bar of the applet with user-defined options
Sound	A sound that can play while the applet is running

Figure 3.2.1: Object Types and Descriptions

3.4 Expressions

3.4.1 Primary Expressions

Primary expressions consist of identifiers, numbers, and strings; boolean operators, logical operators, mathematical operators, and assignment operators.

Expressions are used in assignments to create and modify variables and objects. They are used in loops as test cases for further actions and method calls.

3.4.2 Identifiers

An identifier or name can refer to a variety of things. Methods, variables, and objects are labeled with identifiers. Any identifier is limited to the scope in which it is defined, following Java scooping conventions.

Identifiers are strings beginning with an ALPHA character (letters a-z, A-Z, or _) and containing an unspecified length string of alpha-numeric characters and the underscore character.

An identifier cannot have the same spelling as a keyword or Boolean literal. Identifiers are case sensitive; two identifiers are the same only if they have the same string of ASCII values.

3.4.3 Spatial Delimiters

The special delimiters in *Espresso!* are fairly intuitive. See details and descriptions in table 3.4.1 below.

Delimiter(s)	Name(s)	Description/Usage
( )	Parentheses	Used around parameter lists in a method call or object declaration.
{ }	Braces	Used in algorithmic loops or to facilitate scoping.
[ ]	Brackets	Used to in array declarations or assignments
;	Semicolon	Must be at the end of each statement in the card
:	Range	Used to represent a range of numbers (ex. `1:10`)
,	Comma	Used to separate parameters in a parameter list.
.	Dot	Used to indicate decimal number notation (ex. `5.7`) and in modify object statements to denote parameters of an object (ex. `MyButton.color`)

Table 3.4.1: Spatial Delimiters Description and Usage

3.4.4 Boolean Operators

Boolean expressions are composed of identifiers, numbers or strings. Boolean operators evaluate to 0 or 1 (for details see table 3.4.2 below).

The values that are compared by the Boolean operators must be numbers or strings, or variables that represent a numerical or string value.

When evaluating string comparisons strings with a higher alphabetical precedence evaluate higher. For example, the character 'A' has a higher value than 'Z'. Uppercase letters take a higher precedence than their counterpart lowercase letters, but take a lower precedence than any character that is alphabetically before. For example 'B' has a higher precedence than 'b', but a lower precedence than 'a'. Symbols take the lowest precedence and are ranked in order of their Unicode values.

Symbol	Operator Name	Usage	Description
>	Greater than	A > B	Returns 1 A is greater than B, and zero otherwise.
<	Less than	A < B	Returns 1 if A is less than B, and zero otherwise.
>=	Greater than or equal to	A >= B	Returns 1 A is greater than B or if their values are equivalent, and zero otherwise.
<=	Less than or equal to	A <= B	Returns 1 A is less than B or if their values are equivalent, and zero otherwise.
==	Equal to	A == B	Returns 1 only if A and B are equivalent. Returns zero otherwise.
!=	Not equal to	A != B	Returns 1 only if A and B are not equivalent. Returns zero otherwise.

Table 3.4.2: Boolean Operators and Usage (A and B above are identifiers representing literals, numbers, or strings)

3.4.5 Mathematical Operators

Espresso! supports six basic mathematical operations, in order of precedence: exponents and modular arithmetic, multiplication and division, addition and subtraction.

Mathematical Operators are found in expressions that contain numbers and are formed from ASCII characters. The grammar ensures that the precedence levels will be maintained. Balanced parentheses may be used to explicitly state the order in which an expression must be evaluated. The most internal statements with parentheses will be evaluated first.

Typically the result of a mathematical operation is assigned to a variable or used in a Boolean expression. See table 3.4.3 below for operator details.

Operator	Description	Usage	Precedence Level (1 being highest)
^	Exponent	A ^ B	1
%	Modulo	A % B	1
*	Multiplication	A * B	2
/	Division	A / B	2
+	Addition	A + B	3
-	Subtraction	A - B	3

Table 3.4.3: Mathematical Operators and Usage (A and B above are identifiers representing numbers)

3.4.6 Logical Operators

Logical operators take Boolean expressions as operators. There are three logical operators : **NOT, AND,** and **OR**, denoted '!' , "&&"; and "||" respectively.

NOT has the highest precedence among the operators, and **OR** has the lowest. Logical operators are used in expressions only in loops (if, do, while) and not in assignment statements. Expressions involving logical operators evaluate to 0 or 1.

Expressions involving logical operators are greedy. For example, if the first Boolean expression A in (A && B) evaluates to false, the value of B will not be considered. Likewise if A evaluates to true in (A || B), the value of B will not be considered. !A will evaluate to 1 if A is 0, and will evaluate to 0 if A is not equal to 0.

3.4.7 Assigment Operators

Assignment Operators are found in "assignment" statements. The '=' operator is used to create or replace a variable. The other assignment operators are used to modify existing variables. These operators are used in statements shorthand for those that contain the primary assignment operator as well as a mathematic operation.

For example:

`A += B;` is equivalent to `A = A + B;`

See table 3.4.4 below for operator details.

Operator	Description/Name	Expanded equivalent example
=	Assign value on the right to the identifier on the left	N/A
+=	"plus-equals"	A += B ↔ A = A + B
-=	"minus-equals"	A -= B ↔ A = A - B

*=	“times-equals”	A *= B	↔	A = A * B
/=	“div-equals”	A /= B	↔	A = A / B
%=	“mod-equals”	A %= B	↔	A = A %B

Table 3.4.4: Assignment Operators and Descriptions (A and B above are identifiers representing numbers)

3.5 Statements

Statements are composed of keywords, expressions and delimiters. They create the functionality of the language. Statements in *Espresso!* consist of while and do loops, if statements, break statements, declarations, assignments, method calls, print statements, modify object statements, and paint statements. Nesting of loops is allowed and encouraged in the language. Scoping within the statements is consistent with standard Java scoping conventions. In this section, values in <> represent required fields, and values in bold represent required keywords and punctuation.

3.5.1 While Statement

The while statement, denoted while_stmt, must be of the form:

```
while ( <expression> )
{
  <statement >
}
```

This statement is constructed like a standard while loop in Java. While the expression in the parentheses evaluates to true or 1, the statement in the braces will be executed.

3.5.2 Do Statement

The do statement, denoted do_stmt, must be of the form:

```
do ( <identifier>, <number>) {
    <statement>
}
```

This loop will execute the statement inside the braces the given number of times, storing in identifier the number for that iteration. If number is >= 0, the statement will never execute.

3.5.3 If Statement

The if statement, denoted if_stmt, must be of the form:

```
if ( <expression> ) {
      <statement>
}
[elseif ( expression ) {
      statement
```

```
}] (0 or more times)
else {
        statement
}] (0 or 1 time)
```

The statements, expressions, and delimiters in italics are optional. The if part of the statement is required. There may be zero or more "else if" phrases of the statement, and there may be zero or one "else" phrases of the statement.

3.5.4 Break Statement

The break statement, of the form: **`break;`** will break out of the current do, while, or if statement.

3.5.5 Declaration

A declaration assigns a value or set of values to an identifier of a certain type. *Espresso!* is strongly typed in that an identifier declared as one type cannot be modified to represent a value of a different type. There are several forms of acceptable declarations:

- Number declaration:

```
number <identifier> =  <number or identifier> ;
```

- number array declaration:

```
number [ <number or identifier> ] <identifier> = <number or identifier or range
of numbers>
```

- string declaration:

```
string <identifier> = <string or identifier> ;
```

- string array declaration:

```
string [ <number or identifier> ] <identifier> = <string or number or identifier
or range of numbers> ;
```

This declaration will instantiate all the fields in the array to the string, number, identifier, or range on the right of the assignment. If a range is used, for example

```
string[5] myArray = 1:5
```

each string in myArray will have the values 1, 2, 3, 4, and 5, respectively.

- object declaration:

```
<object> <identifier> = <object description> ;
```

  - an object description, denoted `object_desc` is in the following form:

```
( <parameter list> )
```

- a parameter list is specific to each kind of object described above.

- object array declaration:

```
<object> <identifier> [ <number or identifier> ] = <object array description> ;
```

- an object array description, denoted `object_array_desc` is in the following form:

```
<object> ( <identifier or number or string or range of numbers> )
```

This declaration will not set the parameters of each object in the array. In order to change the parameters from the defaults, this statement must be used in conjunction with the array assignment statement described in 3.5.6.

3.5.6 Assignment

Assignments are in one of the following forms:

```
<identifier> <assignment operator> <mdas operator> ;
<identifier> [ <number or identifier> ] <assignment operator> <mdas operator> ;
```

The first form is associated with a single declaration of an number, string, or object. The second form is associated with an array of numbers, strings, or objects. An mdas operator is an element of the grammar that involves one or more identifiers, numbers, and strings that may include the mathematical operators. mdas stands for the end of the standard mathematical precedence acronym PEMDAS because only the operators multiply, divide, add, subtract and mod may be involved. Assigments are used to modify or replace the value denoted by a previously defined identifier.

3.5.7 Method Call

Methods are user-defined functions that, given some parameters, perform a useful operation. A method is defined with the keyword ‘method’, followed by the function identifier or name, followed by a parenthesized parameter list (which may be empty), followed by the function body in braces. The body of the method should consist of 0 or more statements, followed by a return statement. A return statement consists of the keyword ‘return’, followed optionally by the expression which is to be returned.

For example:

```
method <identifier> ( <parameter list> ) { <method body> }
```

While methods can be used to modify existing variables, an identifier may also be assigned to the method call in a declaration. This will assign the value returned by the method to the identifier.
For example:

```
number x = myMethod(parameter1, parameter2);
```

3.5.8 Print Statement

The print() function takes one or more parameters and prints them one by one to standard output. The parameter type may be string, number, or object. It is in the following form:

```
print <one or more strings> ;
```

3.5.9 Paint Statements

There are three kinds of paint statements: paint, repaint, and setBackground.

- **paint** is in one of the following forms:

  ```
  paint ( <identifier or string or object> );
  paint ( <identifier or string or object> , <number> , <number> ) ;
  ```

  The paint statement takes one or three parameters. The first should be the identifier of the string, number or object which is to be painted to the current card. The next two parameters should be the x and y coordinates corresponding to the position at which to paint. If left out, the default position will be (0, 0). The grid coordinates are structured such that (0, 0) is in the upper left hand corner of the card, with increasing x values to the right, and increasing y values downward.

- The **repaint** statement, in the form: `repaint;` redraws the current card, updating any variables which have changed in value or position.

- The **setBackground** statement sets or resets the background of the current card. It takes one color argument to which the background is to be set in the following form:

  ```
  setBackground ( <string> ) ;
  ```

3.5.10 Modify Object Statement

The modify object statement is used to change the parameters of an existing object, or to initially change the parameters of an array of objects from the default values. It is in the following form:

```
<identifier> . [parameter (optional) ] = <identifier> ;
```

If no parameter is specified, the entire object is changed. The parameter must match an acceptable element of that object's parameter list. For A list of the parameters and their defaults for each object, see section 3.6.

3.5.11 Set Dimension Satement

The setDimension statement sets or resets the dimensions of the current card. It takes two number arguments: width and height. It is in the following form:

```
SetDimension ( <number> , <number> ) ;
```

A call to setDimension will also trigger a call to repaint() in order to ensure that all elements are placed properly within the card.

3.6 Object Construction

3.6.1 Object Parameter Values

Table 3.6.1 below includes the name of the parameters for each kind of object, as well as their default values. Assignment and declaration of objects must either include no parameters (in which case all values will be set at the default), or all parameters. An object modification must use the correct keyword after the dot operator that corresponds to a valid parameter name for the object type labeled with the given identifier.

Object Type	Parameter Keyword	Default	Acceptable values
TextBoxes	WIDTH	200	Any positive number
	HEIGHT	30	Any positive number
	BGCOLOR	White	See section 3.6.2
	FGCOLOR	Black	See section 3.6.2
	JUSTIFICATION	Center	Left, Right, Center
	SIZE	0	-2, -1, 0, 1, 2
	TEXT	Empty	String or Identifier
Buttons	WIDTH	200	Any positive number
	HEIGHT	30	Any positive number
	BGCOLOR	White	See section 3.6.2
	FGCOLOR	Black	See section 3.6.2
	JUSTIFICATION	Center	Left, Right, Center
	SIZE	0	-2, -1, 0, 1, 2
	ACTION	None	None, Card Name (will link to card), Sound, Button (will change to new button)
	TEXT	Empty	String or Identifier
ChkBoxes	SIZE	0	-2, -1, 0, 1, 2
	TEXT	Empty	String or Identifier
	UNIQUE	True	True or False (relevant only for an

			array of checkboxes, indicates whether only one checkbox can be highlighted at a time)
Images	SOURCE	Empty	Source of .jpeg or .gif file
	WIDTH	50	Any positive number
	HEIGHT	50	Any positive number
Shapes	AFOCIX	0	X coordinate of the first foci of an elipse or the center of a circle
	AFOCIY	0	Y coordinate of the first foci of an elipse or the center of a circle
	BFOCIX	0	X coordinate of the second foci of an elipse
	BFOCIY	0	Y coordinate of the second foci of an elipse
	POLYVNUM	4	Number of vertices a polynomial will have – this creates an array of vertex objects that have an x and y coordinate.
	POLYV[i]X	0	Will set the x coordinate of the i'th vertex. There are POLYVNUM spots in the vertex array that can be filled in.
	POLYV[i]Y	0	Will set the y coordinate of the i'th vertex. There are POLYVNUM spots in the vertex array that can be filled in.
	COLOR	White	Fill color see section 3.6.2
List	LENGTH	2	Length of the list. Creates an array of strings of size LENGTH.
	COLOR	Black	Text color see section 3.6.2
	SIZE	0	-2, -1, 0, 1, 2
	TEXT	Empty	Strings or identifiers
ScrlBar	JUSTIFICATION	Right	right or bottom
Menus	LENGTH	2	Length of the list. Creates an array of strings of size LENGTH
	TEXT	Empty	Move, exit, open
Sound	TYPE	Bell	Bell, thud, beep, buzz, file

Table 3.6.1 : Object Parameter default and acceptable values.

3.6.2 Colors

Espresso! supports a number of pre-defined colors:

Color	**RGB in Hexadecimal**
BLACK	000000
DARK_GRAY	616161
GRAY	808080

LIGHT_GRAY	C0C0C0
RED	FF0000
PINK	DB7093
ORANGE	FF8429
YELLOW	FFFF00
GREEN	32CD32
CYAN	00FFFF
BLUE	0000FF
WHITE	FFFFFF

Table 3.6.2 : Colors and their hexadecimal values.