

Espresso Language Reference

This document describes a Java subset called Espresso¹. Correct Espresso programs will compile on a Java compiler, and will have the same semantics as the equivalent Java program (we go over a few exceptions to this later in the document).

Here is an overview of what is in Espresso: it has classes, but no inheritance. Classes have (private) fields and (public) methods. There are no `static` fields or methods (except the required method `main`; we'll describe its peculiarities later in the document). There are no inner classes, anonymous classes, or interfaces. For types, Espresso has only `int`, `boolean`, and any user-defined classes. It has a small set of operators and control structures. It has no arrays or `Strings`. The scoping rules are the same as in Java (for the parts that are in Espresso), but Espresso requires all variable declarations come before the statements in a method. Espresso also disallows method overloading. The only I/O is output of integers or blank lines.

Espresso Lexemes

- The keywords of Espresso are as follows: **boolean, class, else, false, if, int, main, new, null, out, println, public, return, static, this, true, void, while, String, System.**
- Identifiers are defined as follows: they may start with an upper or lower case letter or an underscore, followed by zero or more upper or lower case letters, digits, or underscores. In the grammar that follows identifiers are denoted by `ID`.
- Integer literals in Espresso are the decimal integer literals in Java: that is, they are zero by itself, or sequences of digits that do not have any leading zeroes. (Java uses leading zeroes to express octal literals.) In the grammar that follows, integer literals are denoted by `INT_LITERAL`.
- Rules for forming comments and whitespace are the same as in Java. Note that Java `/* */` form comments do not nest.
- The special other lexemes used in Espresso are as follows:

+	*	-	=	==	!=	!	<	&&
()	[]	{	}	.	,	;

¹ Espresso is derived from the MiniJava subset used in the compiler project from the textbook *Modern Compiler Implementation in Java*, 2nd ed., by Andrew Appel, Cambridge University Press, 2002

Espresso Grammar

The notation used is like BNF, but with the following extensions: Foo^* means Foo repeated zero or more times. $[Foo]$ means Foo is optional. (Beware there are also square brackets and an asterisk in the grammar itself in the rules for *MainClass* and *BinOp*, respectively.) In addition, $Foo,^*$ means zero or more Foo 's separated by commas. I.e., the comma doesn't appear after the last Foo in the list.

```

Program → MainClass ClassDecl*
MainClass → class ID { public static void main ( String [] ID) { Decl* Statement* } }
ClassDecl → class ID { Member* }
Member → Type ID ;
        / Type ID (Formal,*) { Decl* Statement* return Expr ; }
Formal → Type ID
Decl → Type ID ;
Type → int
      / boolean
      / ID
Statement → StmtExpr ;
          / if ( Expr ) Statement [ else Statement ]
          / while ( Expr ) Statement
          / System.out.println ( [ Expr ] ) ;
          / { Statement* }
Expr → ID
      / INT_LITERAL
      / true
      / false
      / null
      / this
      / StmtExpr
      / new ID ( )
      / Expr BinOp Expr
      / ! Expr
      / ( Expr )
BinOp → + / * / - /
      / == / != / < / &&
StmtExpr → ID = Expr
          / Expr . ID ( Expr,*)

```

If you are wondering about *StmtExpr* above, this is one of the differences between Java and C/C++: in Java only *some* expressions are allowed to be used directly as statements – generally the ones with side effects; this does not prohibit other expressions to be used as sub-expressions of the statement. E.g.:

```

a+b; -- not legal in Espresso (or Java)
foo.bar(a+b); -- legal in Espresso (and Java)

```

Espresso semantics

We'll describe a few of the semantic rules of Espresso here, to clarify what is in the subset, as well as mentioning some differences between Java and C++, where you might not be familiar with them. This is not meant to be a complete description: you will definitely need to refer to the Java Language Specification for details on other parts of Espresso (see last section of this document).

One difference between C++ and Java at the statement and expression level is that the condition expression in an `if` or `while` must be type `boolean`. This *prevents* (usually faulty) expressions such as the following to pass a Java (or Espresso) compiler:

```
if (i = 0) . . . // oops, used assignment instead of ==
                // Java compiler will catch this
```

This requirement of Java does not show up in the Espresso grammar given earlier, because type-checking rules are generally outside the scope of what we can express in a grammar.

You may have noticed from the Espresso grammar that there is no accessibility (e.g., `public`, `private`, `protected`) specified for the data fields of a class in Espresso. The default rule in Java for this syntax is package scope. However, because we have no inheritance here, and no syntax to access the field of an object other than “this” (i.e., `ID` is a valid expression, but `ID.ID` is not) fields are effectively private in Espresso.

`System.out.println` is a special-case statement used for compatibility with the Java compiler: note from the grammar that it cannot be used as an expression, unlike other method calls. In Espresso it is only defined to work on integer expressions, or to print a blank line if no argument is given.

The other special-case feature is the `main` method, described in detail in the next section.

main method

For compatibility with Java compilers, the main method has syntax that doesn't appear elsewhere in Espresso. The header for main appears as follows:

```
public static void main( String [] id)
```

We will go through the parts of this that are specific to `main`, one by one to explain them:

- `static` keyword. No other methods are allowed to be `static` (see grammar).
- `void` keyword. This can only appear in the main method definition (see grammar).
- `main` keyword. Since `main` is a keyword, not an identifier, `main` is not allowed to be called from within the program (see grammar for method invocation). Thus, even though there is one static method defined, there are no static method *calls* in

Espresso. (The runtime system will call `main` initially, but this call is not part of the program text.)

- `String [] id`. `String` is a keyword and `[` and `]` are valid symbols in the language. However, there is no other place in an Espresso program that `String`, `[`, and `]` are allowed to be used (see grammar). The formal parameter, `id`, defined in the header for `main`, can never be used in the `main` method because there are no operations or statements defined for `String` values. However, by the scoping rules for Espresso (and Java), it is an error to define another variable with the same name as `id` within `main`.

In addition, `main` has no `return` statement.

Initializing Object Instances

You may have noticed that there are no constructors in Espresso. The semantics of class creation without constructors is the same as in Java, but we'll review it here for clarity. The semantics of doing `new Foo()` on a class `Foo` with no constructors is to dynamically allocate the object, and then initialize all the fields of the object with their default values. The default values for Espresso types are as follows: for `int`, it's zero, for `boolean`, it's `false`, for all user defined class types it's `null` (the last of these are called “reference types” in the Java manual).

Some Incompatibilities with Java

We list here some kinds of Espresso programs that will fail on a Java compiler.

- Any identifier used in an Espresso program that is a Java-but-not-Espresso *keyword* will fail on a Java compiler.
- A requirement of Java is that the compiler check if variables are assigned before use for every path through the program. This is called the Definite Assignment requirement. Here is an example of a code segment that would cause a compile error in the Java compiler by this rule:

```
{
    int i;
    int j;
    j = i + 10; // i has not been initialized
}
```

The techniques necessary to implement this requirement are beyond the scope and time-constraints of this course. So, the above code segment, for example, will succeed on an Espresso compiler. It is an “incorrect” program, in that it will lead to undefined results at run-time, but that will not be detected at compile-time (as is the case with the same code segment in C/C++).

Espresso program files

An Espresso program consists of one file only, with the extension `esp`. As in Java the prefix part of the file name must be the same as the name of the class that contains the main method. See example that follows.

A first Espresso program: `Small.esp`

```
class Small {
    public static void main (String[] args) {
        Pair p;
        p = new Pair();
        p.init(3,5);
        p.print();
    }
}

class Pair {
    int x;
    int y;

    Pair init(int i, int j) {
        x = i;
        y = j;
        return this;
    }

    int print() {
        System.out.println(x);
        System.out.println(y);
        return 0;
    }
}
```

The program above, as well as a longer example program, is available in electronic form on the course web page.

Where to get more information

We aren't intending to give a full language definition here because the complete semantics of Java are defined elsewhere. For any parts of Espresso you are unsure of, please refer to the Java Language Specification, Second Edition, an html version of which is linked from following page: <http://java.sun.com/docs/books/jls/>. It's also linked from the course web page. Another way to find out how or whether something works in Espresso is to feed your sample Espresso program to a Java compiler. We will also be going over some of the trickier parts of the language in lecture and in project-related documents over the course of the semester.