CSC 460/660
Compiler Construction

The Espresso Compiler Project

School of Computer Science
Howard Hughes College of Engineering
University of Nevada, Las Vegas

Chapter 0

Introduction

This document serves two purposes; firstly, it is the project description for the course, and secondly, it is a guide to solving the problems that might arise while writing the compiler.

At first, it looks terrifying because of its size, but do not worry, most of the text is there to make your life easier. It explains what you have to do and how you do it, so do not worry about the size of this document.

0.1 The Espresso Project

You will be implementing a compiler for a language called **Espresso**; **Espresso** is a subset of real Java, and when you have implemented the final phase, you should be able to execute the output from your compiler using the regular JVM installed on the system. The 6 phases are as follows:

- Phase 1 — Using the tools *JFlex* and *Java CUP* construct a scanner and a parser for the **Espresso** language.
- Phase 2 — Add action to the parser to build a syntax tree for the parsed program.
- Phase 3 — Add a symbol table to the project.
- Phase 4 — Perform type checking.
- Phase 5 — Modifier checking.
- Phase 6 — Generate intermediate assembler code, which can be assembled to Java class files using *Jasmin*.

You should implement the project in Java, and each phase will have a number of files that I provide. If you do not get one phase 100% correct, it is OK. Each phase's handout files will contain a working version of the previous phase (i.e., my reference solution). You may choose to build on that or you may chose to fix up you own version and continue working on that.

If you are absolutely against implementing this project in Java you may use C/C++ with Lex and Yacc instead of *JFlex* and *CUP*, but you are 100% on you own, and there will not be any reference implementation or any handouts provided in C/C++.

This project may be done in groups of 2 people. If you insist that you want to do it on your own that is OK as well, but no more than 2 people in a group. If you are taking this course as CSC 660 you **must** work alone and you **must** implement the **Espresso+** language, i.e., the extended **Espresso** language. If you are taking CSC 460 and you implement **Espresso+** you get 5% extra; if you are taking CSC 660 you **must** implement **Espresso+**. If you implement `Espresso*` (Which is **Espresso+** plus arrays) you get another 5% regardless of which course you are registered in. If you are taking this course as CSC 460, and if you implement **Espresso+**

you get 5% extra. If you implement **Espresso***, you get an additional 5% (for both CSC 460 and 660.)

This document is available on the web through the course web page.

0.2 Academic misconduct

Cheating is not allowed. You all know what cheating is and what constitutes cheating. The general rules and regulations from the University of Nevada, the College of Engineering and the School of Computer Science apply; if you are not familiar with these rules you are strongly encouraged to find them and read them as **you** are responsible for knowing what is considered cheating. In addition the following rules apply to this course: You may not share code, you may not use other peoples code, you may no decompile my or other people's code, you may not hand in code that you found on the web. You may **ONLY** hand in code that you wrote yourself without anyone's help except for the help you get from me or the teaching assistant. By signing up for this class (which you will have done by now if you are reading this) you acknowledge that I, as the instructor of this course has the right, at my discretion, to give you 0 on an assignment or fail (give you an F) the course if I have reason to believe that you have cheated. If you do not agree with this statement, please come and see me immediately. If you are caught cheating the punishment can be as severe as expulsion from the university or the college or/and a note on your academic transcript which shows that you cheated.

Please familiarize yourself with the information found at `http://studentconduct.unlv.edu/misconduct/policy.html`.

0.3 Espresso/Espresso+/Espresso*

Espresso+ extends **Espresso** by `case`-statements, the ternary operator `)... ? ... : ...` as well as interfaces and abstract methods. **Espresso*** further extends the language by adding arrays.

0.4 The Handout Code

The handout code can be obtained using the `espressou` script that is available on `java.cs.unlv.edu` in the directory `/home/matt/CSC460/Espresso/`. Download or copy this script file and place it in the directory that you want to serve as your project root directory. Remember to make the file executable (`chmod 755 espresso`). I suggest that you make a directory called `Espresso` in your CSC 460/660 directory, if you have one. For each phase, the `espressou` script will make a subdirectory called `PhaseX` where `X` is the phase. See appendix E for more information about this script.

The installed project will have the following structure. Some of the files will be skeleton files with no implementation; the implementation will follow in subsequent phases. Some of the AST files will be missing in phase 1, but will be provided when phase 2 is installed.

```
Espresso/--+
           + -- espressou                 (the phase install script)
           + -- PhaseX/--+
                         +-- Include       (directory containing Espresso include files)
                         +-- Lib           (directory containing class files for the includes)
                         +-- build.xml     (ant build script)
                         +-- espressoc     (run script to execute your Espresso compiler)
                         +-- espressocr    (run script to execute the Espresso reference compiler)
                         +-- id.txt        (id file - do not forget to fill it in)
+-- src/-----+
             +-- AST/
             +-- CodeGenerator/
             +-- Espressoc.java
             +-- Instruction/
             +-- Jasmin/
             +-- ModifierChecker/
             +-- NameChecker/
             +-- Parser/
             +-- Phases/
             +-- Scanner/
             +-- TypeChecker/
             +-- Utilities/
```

Each phase has the same structur, but might have a different number of files in each directory.

0.5 Compiling your Project

After you have installed the project, you can simply issue the command `ant` in the directory in which the `build.xml` file is located. That will compile the entire project. If you make any changes in the cup or the jflex file you should rebuild using the same command, but if you only make changes in java files, you can skip the generation of the scanner and the parser by issuing the `ant espresso` command instead; that will simply compile the changed java files.

0.6 Running the Reference Implementation

In order to check your compiler against my version of the **Espresso** compiler, you can use the `espressocr` script. That will submit your file to my reference implementation and return the output.

0.7 Submitting Your Project

To submit your project you should use the computer science grading and file management system at `http://csgfms.cs.unlv.edu`. You can submit your project as many times as you want, as long as it is before the deadline. Only the most recent submission will be graded, unless you tell us otherwise. Make sure you use the `espressou` script to build a `.tar` file to hand in: `espressou tar <phase number>`; this should generate a tar file in your directory. This is the file you should upload. **Do not rename it or in any other way change it**. Before you submit it make sure to fill in your name and student number and your email address in the file `id.txt`. The project **must** compile on the CS (`java.cs.unlv.edu`) system too.

0.8 Possible Extensions to Espresso

Should you be interested, it is possible to do a directed studies project that adds more phases to the project, namely an in-memory code generator, which does not write the code to a file, but to an array in memory, and then performs code analysis (builds a Control Flow Graph), and performs code optimization. Should you be interested in working on this please come and see me.

Chapter 1

Phase 1—Scanning and Parsing

1.1 Introduction

Phase 1 of the **Espresso** compiler project is to build a lexical analyzer (also called a scanner or lexer) using *JFlex*, and a parser using *Java CUP*. Your scanner should be able to read any valid **Espresso** program, and your parser should attempt to verify that the input **Espresso** program is correct. If it is incorrect, it should report the position in the input file where the error first occurred. Only report the first error, and do not attempt to implement errors correction.

Espresso is a true subset of Java, so if you know Java you are off to a good start. However you might still want to consult the specifications for the Java language, in particular Chapter 3[1] and Chapter 19[2] and Appendix B. These are both chapters in the The Java Language Specification[3].

1.2 Scanning

The objective is for you to learn to write a scanner using *JFlex*. Remember that the job of a scanner is to 'chop' the input (in our case, **Espresso** programs) into smaller bites, called tokens. For example, if we consider the following small **Espresso** program:

```
public class myClass {
  int field;
  void foo() {
  }
}
```

will be tokenized into the following token stream: $\boxed{\texttt{public}}$ $\boxed{\texttt{class}}$ $\boxed{\texttt{myClass}}$ $\boxed{\{}$ $\boxed{\texttt{int}}$ $\boxed{\texttt{field}}$ $\boxed{\texttt{;}}$ $\boxed{\texttt{void}}$ $\boxed{\texttt{foo}}$ $\boxed{\texttt{(}}$ $\boxed{\texttt{)}}$ $\boxed{\{}$ $\boxed{\}}$ $\boxed{\}}$.

The full definition of the Java tokens are given in Chapter 3[4] of The Java Language Specification[5]. **Espresso** is a strict subset of this specification, and you should be working from our specification of the **Espresso** grammar given in section 1.12. Make sure that you understand all of this before starting on this phase of the project. Also, look at the *JFlex* example code[6], and the on-line manual page for *JFlex* which can be found on the course web page. The simple example is close to what you need to write for this project.

[1] `http://web.cs.unlv.edu/CSC460/docs/draft5.2-html/3.doc.html`
[2] `http://web.cs.unlv.edu/CSC460/docs/draft5.2-html/19.doc.html`
[3] `http://web.cs.unlv.edu/CSC460/docs/draft5.2-html/j.title.doc.html`
[4] `http://web.cs.unlv.edu/CSC460/docs/draft5.2-html/3.doc.html`
[5] `http://web.cs.unlv.edu/CSC460/docs/draft5.2-html/j.title.doc.html`
[6] `/home/CSC460/Java-Utils/JFlex/examples/`

Your scanner must report any lexical errors that occur, as well as the line number. You only need to report the first error that you encounter, and then abort the scanner. For every token that is correctly scanned, print the token name, the lexeme, and the line and column position at which it was detected. See the section about tokens.

1.2.1 The theory of lexical analysis

As stated in the previous section, the first part of this phase is to write a scanner specification file, that is, a file which describes the structure of the tokens that make up the Espresso languages, and which can be used as input to *JFlex*, which in turn generates a scanner.

Before we get that far let us just review a little theory. A scanner, or a tokenizer as it is also known, can be implemented using a Deterministic Finite Automata (DFA); remember, a regular expression can be recognized by a DFA, where as a context free grammar (CFG) requires a push-down automata. In other words, that means that a all token (keywords, variable names, special characters, etc.) of a language can be specified using a regular expression, and recognized using a DFA. However, writing a complete scanner by hand can be a little tedious and error prone, and why bother when we have perfectly good tools that read a specification of our tokens and generates a Java class that will do the scanning for us. (*JFlex* generates a Java class canned `Scanner.java`, which has a method called *next_token()*, which the parser can call to have the next token scanned from the input.

1.3 A Flex File

A Flex file is divided into 3 section (separated by the `%%` symbol):

```
UserCode
%%
Options and declarations
%%
Lexical rules
```

You will be focusing on the 'Lexical rules' section only. I have filled in the other parts already. If you want to know more about the other sections you should consult the JFlex User's Manual. I have provided a few examples of how to enter the lexical rules in the `espresso.flex` file.

1.3.1 UserCode

This section is left empty for this project.

1.3.2 Options and Declarations

This part has 3 different types of declarations.

The first part looks like this:

```
%class Scanner
%7bit
%pack
%cup
%line
%column
```

which instructs the scanner generator (the *JFlex* program) to generate the scanner in a class class *Scanner* and to use 7-bit packed character representation. The `%cup` line tells the scanner generator to generate code that interfaces with a parser generated by the *Cup* tool, and finally, the last two lines tell the scanner generator to make available line and column information for all token scanned.

Following this is a block of actual Java code:

```
%{
  public static String curLine = "";
  public static int lineCount = 0;
  public static boolean debug = false;

  public void addToLine(String s, int line) {
    if (line != lineCount)
      curLine = s;
    else
      curLine = curLine + s;
    lineCount = line;
  }


 private java_cup.runtime.Symbol token(int kind) {
    Token t;
    addToLine(yytext(), yyline+1);
    t = new Token(kind, yytext(), yyline+1, yycolumn+1, yycolumn + yylength());
    if (debug)
      System.out.println(t);
    return new java_cup.runtime.Symbol(kind, t);
  }
%}
```

The first method (*addToLine()* is used to keep track of what we have read in the current line; this comes in handy when we encounter syntax errors. The second method (*token*) is the method that the scanner will call when it has recognized a new token. The *kind* parameter takes it values from the file `sym.java`, which is generated by the parser (this is what links the scanner and the parser together in the correct fashion).

The third part of this second section is where we define character classes. If you look closely you will see that all classes are defined as regular expressions. For example:

```
FLit1    = [0-9]+ \. [0-9]* {Exponent}?
Exponent = [eE] [+\-]? [0-9]+
```

where an `Exponent` is **either** `e` or `E` followed by an optional `+` or `-` (The `-` is escaped with a backslash because `-` can be used in a range as well), followed by 1 or more `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`. `E-865` and `e67` are both examples of character strings that match `Exponent`. An example of a valid `FLit1` could be `05595.4344E-54`. Try to convince yourself what a general `Flit1` looks like.

1.3.3 Lexical Rules

This section consists of a number of lines of the form:

```
regular-expression { Java action code }
```

where `regular-expression` matches a keyword, symbol, variable name, etc from the language, and `Java action code` is what we want the scanner to do; typically that is to generate a token and return it to the parser. In some situations we might throw an error and terminate.

1.4 Parsing

As we have seen, the scanner tokenizes the input, i.e., chops it up into smaller parts. These smaller parts, the tokens, are passed to the parser, whose job it is to make sure that the token stream makes sense. Makes sense, in this sense (!), means that the token stream describes a syntactically correct program. This is one of the two major jobs of the parser, the other, also very important job of the parser, is to create a parse tree. The parse tree is needed in later phases where we will be doing a lot of checking on the program that is being compiled. Many checks are more easily performed using the parse tree. Phase 2 will deal with constructing the parse tree, so don't worry about that right now.

You need to write a parser specification, also known as a *CUP* file. Take a look at the example found in `/home/CSC460/Java-Utils/java_cup/simple_calc/`. I will give you a skeleton *CUP* file that you can fill in. This file is found in the `PParser` directory after you install phase 1. Study the example well, and also look at the *Java CUP* online manual page, which you can reach from the course web page. Translating the BNF (as given here in this handout and on the course web page to *CUP*-format is a fairly trivial task. However, make sure you are very precise, and do not forget that all *CUP* rules are terminated by a semicolon (;). A BNF rule like `A : B |` ϵ translates into `A ::= B | ;`

Note, that the ϵ is omitted, the 'empty' transition is simply the empty space between the vertical bar and the semicolon that terminates the rule. You can put a comment line `/* empty */` where you see ϵ productions to remind yourself that there is an empty rule.

1.5 Tokens

I have created a class named *Token* that can be used to represent each token that you encounter. Each token stores the string of characters that was recognized as well as the line number and column position at which it was encountered.

The `Token.java` file contains a constructor for constructing objects of the *Token* class. This is done by the lexer, i.e., from the `espresso.flex` file. It also contains a *toString()* method, which is useful for printing out the value of a token.

The `sym.java` file is created by *CUP*, and it contains a class called *sym*, which has a number of constants; as a matter of fact, it is created by *CUP* based on the terminals specified in the *CUP* file. Here is an excerpt of the `sym.java` file:

```
/** CUP generated class containing symbol constants. */
public class sym {
  /* terminals */
  public static final int SHORT = 4;
  public static final int IDENTIFIER = 38;
  public static final int ANDEQ = 78;
  public static final int GT = 46;
  ...
  ...
```

The rule for the `>` symbol in the `espresso.flex` file thus looks like this:

```
">"                              { return token(sym.GT); }
```

This links the `>` symbol with the `GT` "name". This name originated in the *CUP* file in the following manner:

```
terminal Token EQ, LT, GT;      // = < >
```

which defines 3 terminals, `EQ`, `LT`, and `GT`, all of type *Token*. This is why the scanner action returns a new *Token* object.

The tricky part is that even though your parser (generated by *CUP*) uses the lexer generated by *JFlex*, your flex file uses `sym.java` to type tokens.

The string array *names[ ]*, found in `Token.java`, contains a string representation of all the tokens; this is useful for the *toString()* procedure. This is the only way you can get a textual representation of the token type, as the type is stored as an integer (defined in `sym.java`). This relationship between *JFlex*, *CUP* and your own code is a little difficult to understand, but if you look at the *names[ ]* array in `Token.java`, the first two entries 0 and 1 (EOF and error) are reserved position. The following positions must match the order in which you define the terminals in the *CUP* file. So for each terminal in the *CUP* file, you must add a new entry in the *names[ ]* array in `Token.java`. As you can see, `GT` is just a constant for the value 46, and if you look in the `Token.java` file, this value is stored in the variable *sym*, which is used as an index into the *names[ ]* array; this is why the order of this array matters, it must be the same as the order of the terminals in the *CUP* file. Don't forget this!

In the flex file, when you create a new token, you pass a type, and this type always has the form *sym.XXX* where *XXX* is the name of the terminal that you recognized.

So in short, make sure that when you define terminals in your *CUP* file that you do it in the order similar to the order of the *name[ ]* array in the `Token.java` file, else things won't look right. You may (or should) use the names for the terminals that are matching to those specified in the *names[ ]* array, i.e., the terminal `'--'` is named `MINUSMINUS`, so use that name in your *CUP* file as well.

1.6 Work List

You are of course free to do things in which ever order you like, but I suggest the following approach.

- Read through the grammar.
- Using the grammar rules, pick out all the terminals and compile a list of them, in the right order, with their names.
- Finish the flex file by adding in all the missing terminals; use the examples already given in the file to get the syntax correct.
- Add all terminals to the *CUP* file in the right order, and remember they are all of type *Token*.
- Add all the non-terminals to the *CUP* file. Since we are not building a parse tree in this phase, simply use the type *AST* as the type of all the non-terminals. We will refine this when we start building the parse tree in the next phase.
- Add the productions according to the grammar (see below how to interpret the grammar). Do not construct any parse tree at this time. You may leave the action part of the production, i.e., the part that appears between `{: :}` blank. For debugging purposes, you could also put print statements that print out the names (and values) associated with the productions. This makes it easier to trace any potential errors. Be VERY careful when you transcribe the grammar; the slightest mistake can cause many hundreds of shift/reduce and reduce/reduce conflicts. I suggest you do it in an iterative fashion, slowly adding more and more information to the file. Do not type in the grammar by hand—copy it from the web page and then 'fix it up'. This will save you a lot of time and headaches caused by mistyping. Also be careful about semicolons; (!) all *CUP* productions are terminated by a semicolon. The grammar contains ';' as a terminal symbol, but in *CUP* that would be represented by the terminal SEMICOLON (this is what the semicolon is

called by *Flex*. If you miss these SEMICOLON terminals you will get a lot of shift/reduce and reduce/reduce conflicts, so look for these as the first thing you do if you have such error messages.

- Compile the compiler using the `ant` command.
- Run the 'espressoc' script to test your compiler.
- Compare to my reference implementation.

1.7 Test Files

You should install the tests files into your project. This is most easily done using the `espressou` script. To install the tests in a directory called `Tests` in the same directory as the `espressou` script simply do a `./espressou install tests .` (the `.` is part of the command!)

To determine whether your compiler is performing correctly, check it against the output from my compiler which you can execute using the `espressocr` script.

1.8 Obtaining the Handout Code

In the directory where you placed the `espressou` script, run the following command to install and unpack the handout code for phase 1: `./espressou install 1`. The script will ask for your password to `java.cs.unlv.edu` if you are not that specific machine. The handout code contains `jar` files for both the *Java CUP* and the *JFlex* utilities, so you should not have any problems compiling your project on any machine that has a Java compiler installed.

1.9 Running the Reference Implementation

In order to check your compiler against my version of the **Espresso** compiler, you can use the `espressocr` script. That will submit your file to my reference implementation and return the output.

1.10 Pre-supplied Code and Testing

I have provided a skeleton of the scanner and the parser which has solved the problems of getting *Java CUP* to talk to the scanner produced by *JFlex*. You still have to finish the flex and the *CUP* file.

I also provide the skeleton of the files that define the abstract syntax tree nodes. Since we are not building a parse tree you don't need those until next time, so for now I have only supplied `AST.Java` to assure that the return type of all the non-terminals is available.

When you hand in your project we will use the `build.xml` and `ant` to compile your project, and the `espressoc` script to run your compiler. If we cannot do that we will not mark your project, i.e., you get 0 on this phase. We will do the following to test your compiler:

```
ant clean
ant
espressoc <testfile.java>
```

so please make sure this process works on `java.cs.unlv.edu`

1.11 Due Date

The first phase of the project is due ______________________________ before midnight.

1.12 The Espresso and Espresso+ Grammar

The following is a BNF description of the grammar for **Espresso**. The parts of the grammar in *italics* are the **Espresso+** extensions. The underlined parts are the parts that are needed to implements arrays. You can add them now if you like, or you can wait until phase 7. Even if you do not think that you want to implement phases 2, 3, 4 and 5 for **Espresso+**, I recommend that you at least do it for phase 1. The amount of extra work you have to do is neglectable. Do not manually type in this grammar, copy it from the website where the assignment is posted as HTML.

This table might help you read the grammar:

Type face	Explanation
regular font	non-terminals
ALL CAPS	terminal literals
boldface	terminals (keywords etc.)
italics	**Espresso+**
underline	**Espresso***
|	Choice
ϵ	Empty production

The Syntactic Grammar

goal	:	compilation_unit

Lexical Structure

literal	:	INTEGER LITERAL
	|	LONG LITERAL
	|	FLOAT LITERAL
	|	DOUBLE LITERAL
	|	BOOLEAN LITERAL
	|	STRING LITERAL
	|	NULL LITERAL
	|	CHARACTER LITERAL

Types, Values, and Variables

type	:	primitive_type
	|	reference_type
primitive_type	:	**boolean**
	|	**byte**
	|	**short**
	|	**int**
	|	**long**
	|	**float**
	|	**double**
	|	**char**
	|	**String**
reference_type	:	class_or_interface_type
	|	array_type
class_or_interface_type	:	name

class_type	:	class_or_interface_type
interface_type	:	*class_or_interface_type*
array_type	:	primitive_type dims
	|	name dims

Names

name	:	IDENTIFIER

Packages

compilation_unit	:	import_declarations_opt type_declarations_opt
import_declarations_opt	:	import_declarations
	|	ϵ
import_declarations	:	import_declaration
	|	import_declarations import_declaration
import_declaration	:	**import** name **;**
type_declarations_opt	:	type_declarations
	|	ϵ
type_declarations	:	type_declaration
	|	type_declarations type_declaration
type_declaration	:	class_declaration
	|	interface_declaration

Productions for LALR(1) grammar

modifiers_opt	:	modifiers
	|	ϵ
modifiers	:	modifier
	|	modifiers modifier
modifier	:	**public**
	|	**private**
	|	**static**
	|	**final**
	|	***abstract***

Classes
Class Declaration

class_declaration	:	modifiers_opt **class** name super_opt *interfaces_opt* class_body
super_opt	:	super
	|	ϵ
super	:	**extends** class_type
interfaces_opt	:	*interfaces*
	|	ϵ

interfaces	:	***implements*** *interface_type_list*
interface_type_list	:	*interface_type*
	|	*interface_type_list* **,** *interface_type*
class_body	:	**{** class_body_declarations_opt **}**
class_body_declarations_opt	:	class_body_declarations
	|	ϵ
class_body_declarations	:	class_body_declaration
	|	class_body_declarations class_body_declaration
class_body_declaration	:	field_declaration
	|	method_declaration
	|	static_initializer
	|	constructor_declaration

Field Declarations

field_declaration	:	modifiers_opt type variable_declarators **;**
variable_declarators	:	variable_declarator
	|	variable_declarators **,** variable_declarator
variable_declarator	:	variable_declarator_id
	|	variable_declarator_id **=** variable_initializer
variable_declarator_id	:	name
	|	variable_declarator_id [ ]
variable_initializer	:	expression
	|	array_initializer

Method Declarations

method_declaration	:	modifiers_opt type name **(** formal_parameter_list_opt **)** method_body
	|	modifiers_opt **void** name **(** formal_parameter_list_opt **)** method_body
method_body	:	block
	|	**;**
formal_parameter_list_opt	:	formal_parameter_list
	|	ϵ
formal_parameter_list	:	formal_parameter
	|	formal_parameter_list **,** formal_parameter
formal_parameter	:	type variable_declarator_id

Static Initializers

static_initializer	:	**static** block

Constructor Declarations

constructor_declaration	:	modifiers_opt name **(** formal_parameter_list_opt **)** constructor_body

constructor_body	:	**{** explicit_constructor_invocation block_statements **}**
	|	**{** explicit_constructor_invocation **}**
	|	**{** block_statements **}**
	|	**{ }**
explicit_constructor_invocation		
	:	**super (** argument_list_opt **) ;**
	|	**this (** argument_list_opt **) ;**

Interfaces

Interface Declarations

interface_declaration	:	*modifiers_opt* ***interface*** *name extends_interfaces_opt interface_body*
extends_interfaces_opt	:	*extends_interfaces*
	|	ϵ
extends_interfaces	:	***extends*** *interface_type*
	|	*extends_interfaces* **,** *interface_type*
interface_body	:	**{** *interface_member_declarations_opt* **}**
interface_member_declarations_opt	:	*interface_member_declarations*
	|	ϵ
interface_member_declarations	:	*interface_member_declaration*
	|	*interface_member_declarations interface_member_declaration*
interface_member_declaration	:	*constant_declaration*
	|	*abstract_method_declaration*
constant_declaration	:	*field_declaration*
abstract_method_declaration	:	*modifiers_opt type name* ***(*** *formal_parameter_list_opt* ***) ;***
	|	*modifiers_opt* ***void*** *name* ***(*** *formal_parameter_list_opt* ***) ;***

Arrays

array_initializer	:	**{** variable_initializers_opt **}**
variable_initializers_opt	:	variable_initializers
	|	ϵ
variable_initializers	:	variable_initializer
	|	variable_initializers **,** variable_initializer

Blocks and Statements

block	:	**{** block_statements_opt **}**
block_statements_opt	:	block_statements
	|	ϵ
block_statements	:	block_statement
	|	block_statements block_statement
block_statement	:	local_variable_declaration **;**
	|	statement

local_variable_declaration	:	type variable_declarators
statement	:	statement_without_trailing_substatement
	|	if_then_statement
	|	if_then_else_statement
	|	while_statement
	|	for_statement
statement_no_short_if	:	statement_without_trailing_substatement
	|	if_then_else_statement_no_short_if
	|	while_statement_no_short_if
	|	for_statement_no_short_if
statement_without_trailing_substatement	:	block
	|	empty_statement
	|	expression_statement
	|	*switch_statement*
	|	do_statement
	|	break_statement
	|	continue_statement
	|	return_statement
empty_statement	:	**;**
expression_statement	:	statement_expression **;**
statement_expression	:	assignment
	|	pre_increment_expression
	|	pre_decrement_expression
	|	post_increment_expression
	|	post_decrement_expression
	|	method_invocation
	|	class_instance_creation_expression
if_then_statement	:	**if (** expression **)** statement
if_then_else_statement	:	**if (** expression **)** statement_no_short_if **else** statement
if_then_else_statement_no_short_if	:	**if (** expression **)** statement_no_short_if **else** statement_no_short_if
do_statement	:	**do** statement **while (** expression **) ;**
switch_statement	:	***switch (*** *expression* ***)*** *switch_block*
switch_block	:	**{** *switch_block_statement_groups_opt* **}**
switch_block_statement_groups_opt	:	*switch_block_statement_groups*
	|	ϵ
switch_block_statement_groups	:	*switch_block_statement_group*
	|	*switch_block_statement_groups switch_block_statement_group*
switch_block_statement_group	:	*switch_labels block_statements*
switch_labels	:	*switch_label*
	|	*switch_labels switch_label*
switch_label	:	***case*** *constant_expression* ***:***
	|	***default :***
while_statement	:	**while (** expression **)** statement

while_statement_no_short_if	:	**while** (expression) statement_no_short_if
for_statement	:	**for** (for_init_opt **;** expression_opt **;** for_update_opt) statement
for_statement_no_short_if	:	**for** (for_init_opt **;** expression_opt **;** for_update_opt) statement_no_short_if
for_init_opt	:	for_init
	|	ϵ
for_init	:	statement_expression_list
	|	local_variable_declaration
for_update_opt	:	for_update
	|	ϵ
for_update	:	statement_expression_list
statement_expression_list	:	statement_expression
	|	statement_expression_list **,** statement_expression
break_statement	:	**break ;**
continue_statement	:	**continue ;**
return_statement	:	**return** expression_opt **;**

Expressions

primary	:	primary_no_new_array
	|	array_creation_expression
primary_no_new_array	:	literal
	|	**this**
	|	(expression)
	|	class_instance_creation_expression
	|	field_access
	|	method_invocation
	|	array_access
array_creation_expression	:	**new** primitive_type dim_exprs dims_opt
	|	**new** class_or_interface_type dim_exprs dims_opt
	|	**new** primitive_type dims array_initializer
	|	**new** class_or_interface_type dims array_initializer
dim_exprs	:	dim_expr
	|	dim_exprs dim_expr
dim_expr	:	[expression]
dims_opt	:	dims
	|	ϵ
dims	:	[ ]
	|	dims [ ]
class_instance_creation_expression	:	**new** class_type (argument_list_opt)
argument_list_opt	:	argument_list
	|	ϵ

```
argument_list                  :  expression
                               |  argument_list , expression

field_access                   :  primary . name
                               |  super . name
                               |  name . name

method_invocation              :  name ( argument_list_opt )
                               |  primary . name ( argument_list_opt )
                               |  super . name ( argument_list_opt )
                               |  name . name ( argument_list_opt )

array_access                   :  name [ expression ]
                               |  primary_no_new_array [ expression ]

postfix_expression             :  primary
                               |  post_increment_expression
                               |  post_decrement_expression
                               |  name

post_increment_expression      :  postfix_expression ++

post_decrement_expression      :  postfix_expression − −

unary_expression               :  pre_increment_expression
                               |  pre_decrement_expression
                               |  + unary_expression
                               |  - unary_expression
                               |  unary_expression_not_plus_minus

pre_increment_expression       :  ++ unary_expression

pre_decrement_expression       :  − − unary_expression

unary_expression_not_plus_minus : postfix_expression
                               |  ~ unary_expression
                               |  ! unary_expression
                               |  cast_expression

cast_expression                :  LPAREN expression RPAREN unary_expression_not_plus_minus
                               |  LPAREN primitive_type RPAREN unary_expression

multiplicative_expression      :  unary_expression
                               |  multiplicative_expression * unary_expression
                               |  multiplicative_expression / unary_expression
                               |  multiplicative_expression % unary_expression

additive_expression            :  multiplicative_expression
                               |  additive_expression + multiplicative_expression
                               |  additive_expression − multiplicative_expression

shift_expression               :  additive_expression
                               |  shift_expression << additive_expression
                               |  shift_expression >> additive_expression
                               |  shift_expression >>> additive_expression

relational_expression          :  shift_expression
                               |  relational_expression < shift_expression
                               |  relational_expression > shift_expression
                               |  relational_expression <= shift_expression
                               |  relational_expression >= shift_expression
```

```
                              |  relational_expression instanceof name

equality_expression           :  relational_expression
                              |  equality_expression == relational_expression
                              |  equality_expression != relational_expression

and_expression                :  equality_expression
                              |  and_expression & equality_expression

exclusive_or_expression       :  and_expression
                              |  exclusive_or_expression ˆ and_expression

inclusive_or_expression       :  exclusive_or_expression
                              |  inclusive_or_expression | exclusive_or_expression

conditional_and_expression    :  inclusive_or_expression
                              |  conditional_and_expression && inclusive_or_expression

conditional_or_expression     :  conditional_and_expression
                              |  conditional_or_expression | | conditional_and_expression

conditional_expression        :  conditional_or_expression
                              |  conditional_or_expression ? expression : conditional_expression

assignment_expression         :  conditional_expression
                              |  assignment

assignment                    :  left_hand_side assignment_operator assignment_expression

assignment_operator           :  =
                              |  *=
                              |  /=
                              |  %=
                              |  +=
                              |  -=
                              |  <<=
                              |  >>=
                              |  >>>=
                              |  &=
                              |  ˆ=
                              |  |=

left_hand_side                :  name
                              |  field_access
                              |  array_access

expression_opt                :  expression
                              |  ϵ

expression                    :  assignment_expression

constant_expression           :  expression
```

Chapter 2

Phase 2—Parse Tree Construction

2.1 Introduction

The second phase of the **Espresso** compiler project is concerned with adding actions to the *CUP* file. These actions will build a parse tree for the program being parsed. We need such a parse tree in the next 4 phases. Many checks and computations can be performed by implementing traversals of the parse tree.

If the input is syntactically correct, your compiler should build a parse tree. This is done by providing Java code to construct the tree between the `{:  ...  :}` symbols.

2.2 Modifying the CUP file

As you probably recall from the first phase, you either left the actions blank or added some print statements for debugging purposes. You will now have to fill in the real code to build the parse tree. A parse tree will consist of a number of different nodes, depending on which construction each node represents. Examples include a node for if-statements, a node for assignments and so on.

I have provided you with a number of different parse tree node classes. All these classes can be found in the handout directory, and you will see they are all sub-classes of `AST.java`, which is the super class for all parse tree nodes (AST = Abstract Syntax Tree).

You **need** to look through these many different classes and figure out which node types should be associated with which action/rules. For example, if you have parsed an if-statement, it is probably a good idea to create a parse tree node of the *IfStat* class. The abstract grammar later in this document will help you determining the correct parse tree node to construct.

Once you have determined which nodes to create at what places in the *CUP* file, you need to fill in the actions. Keep in mind that EVERY production must do something, that is, if you ever leave an action empty in a production, then *CUP* will take that as NULL, which means the empty tree, and anything 'below' that point in the parsing will be removed from the final parse tree; so sometimes if you are not creating a new node, you might need to return the value of one of the right-hand-side terminals or non-terminals.

The way you 'return' something from an action as follows:

```
  {: RESULT = new ParseTreeNode( ... ); :}
```

where `ParseTreeNode` is the name of a parse tree node class. By assigning the *CUP* variable `RESULT` you 'return' something to the previous production. Let us look at the BNF rule for the if-statement.

if_then_statement: **if** **(** expression **)** statement

This was transcribed into *CUP* in the following way:

```
if_then_statement ::= IF LPAREN expression RPAREN statement {: :}
                    ;
```

To build a parse tree node for this if statement, we need to create something that has two children, namely one for the expression and one for the statement. If we look in the `IfStat.java` file, we see the following constructor:

```
public IfStat(Expression expr, Statement thenpart) {
  super(expr);
  nchildren = 3;
  children = new AST[] { expr, thenpart, null };
}
```

This means that we can create an *IfStat* object with the call `new IfStat(...,...);`, however, we need to provide 2 arguments, the first should be an object of the *Expression* class and the second should be an object of the *Statement* class. Naturally we get those from the right hand side of the if_then_statement production. In order to 'get your hands on' these values we need to add some names to the terminals/non-terminals that we are interested in referring to within the actions. Any name can be used, and you simply add it after the terminal/non-terminal in the following way (let us denote the expression `e` and the statement `s`):

```
if_then_statement ::= IF LPAREN expression:e RPAREN statement:s {: :}
                    ;
```

As you can see, the `:e` and the `:s` have been added to the rule. Now we can use these names to refer to the two sub-parse trees we get back from parsing the expression and the statement and build the *IfStat* object representing the parse tree node for the if-statement. We do that by adding this action line to the rule:

```
{: RESULT = new IfStat(e, s); :}
```

This creates a new *IfStat* node with the two children: the expression and the statement.

A special 'list' node called *Sequence* has also been provided. This node is good when you need a number of nodes in sequence, like statements within a block or parameters of a method declaration or invocation.

2.3 The Espresso, Espresso+ and Espresso* Abstract Grammar

The abstract grammar of a language often gives a much better idea of which parse tree nodes to use in which cases. The following is my attempt to turn the BNF grammar into an abstract grammar—it also gives a much better overview of the what a program written in **Espresso** looks like. One of the reasons that the abstract grammar is so much smaller is of course because there are no precedence rules (I have also left out a few other, minor things, like commas separating expressions in function calls etc). In the abstract grammar all binary expressions are represented by one rule, which makes it much easier to read. The names following the `//` in each line is the corresponding parse tree node that the construct creates. Note, the abstract grammar is written in EBNF; `+` means at least one, and `*` means 0 or more repetitions.

```
Compilation ::= Imports ClassInterfaceDecls                    // Compilation

ClassInterfaceDecls ::= (ClassDecl|InterfaceDecl)*             // Sequence

ClassDecl ::= Modifiers "class" Name [ "extends" Name ]        // ClassDecl
              "{" ClassBodyDecl "}"

ClassBodyDecl ::=  Type Var                                    // FieldDecl
              |    Type Name "(" FormalParameterList ")" Block // MethodDecl
              |    Name "(" FormalParameterList ")" Block      // ConstructorDecl
              |    "static" Block                              // StaticInitializerDecl

InterfaceDecl::= Modifiers "interface" Name
                       ["extends" Name+]
                       "{" InterfaceBodyDeclarations "}"       // ClassDecl

InterfaceBodyDeclarations ::= Type Var ";"                     // FieldDecl
                      |   Modifiers Type Name "(" FormalParameterList ")" ";"  // MethodDecl


Modifiers ::= ("abstract"|"public"| ... )*                     // Modifiers


Type ::=        ("byte"|"short"|"int"|"long"|
                 "float"|"double"|"void"
                 "char"|"String")                              // PrimitiveType
    |           Name                                           // ClassType or InterfaceType
    |           Type ("[" "]")+                                // Array Type

Name ::=  IDENTIFIER                                           // Name

Var ::= Name ["=" Expression]                                  // Var

FormalParameterList ::= (Type Name)*                           // Sequence

Block ::=  "{" BlockStatement* "}"                             // Block

Statement ::=
        Type Var ":"                                           // LocalDecl
        Block                                                  // Block
        Expression ";"                                         // ExprStat

        ("this"|"super") "(" ArgumentList ")" ";"
                                                               // CInvocation
        "if" "(" Expression ")" Statement                      // IfStat (else=null)
        "if" "(" Expression ")" Statement
             "else" Statement                                  // IfStat
        "while" "(" Expression ")"  Statement                  // WhileStat
        "for" "(" [StatementList] ";"
                  [Expression] ";"
                  [ExpressionList] ")"
             Statement                                         // ForStat
        "do" "{" StatementList "}" "while"
            "(" Expression ")"                                 // DoStat
        "switch" "(" Expression ")" "{"
            ( ("case" Expression)|("default")
              ":" StatementList )* "}"                         // SwitchStmt
        "break"  ";"                                           // BreakStat
        "continue" ";"                                         // ContinueStat
        "return" [Expression] ";"                              // ReturnStat
        ";"                                                    // null (= empytstatement )

// Note: The range of expressions/statements that can occur in
// the for statement's StatementList and ExpressionList is limited
// by the parser (but we don't care in the abstract grammar :-)

ExpressionList ::= Expression*                                  // Sequence
StatementList  ::= Statement*                                   // Sequence
```

```
Expression ::=
        Name                                                    // NameExpr
        LITERAL                                                 // Literal
        "this"                                                  // This
        "new" ClassType "(" ArgumentList ")"                    // New
        Expression "." Name                                     // FieldRef
        "super" "." name                                        // FieldRef
        Expression "." Name "(" ArgumentList ")"                // Invocation
        Name "(" ArgumentList ")"                               // Invocation
        "super" "." "(" ArgumentList ")"                        // Invocation
        Expression PostOp                                       // UnaryPostExpr
        PreOp Expression                                        // UnaryPreExpr
        Expression BinOp Expression                             // BinaryExpr
        (Name|FieldRef) AssignmentOp Expression                 // Assignment
        "(" (Expression|Type ")" Expression                     // CastExpr
        Expression "instanceof" Name                            // BinaryExpr
        Expression "?" Expression ":" Expression                // Ternary
        Expression ( "[" Expression "]" )+                      // ArrayAccessExpr
        "new" Type ( "[" Expression "]" )+ ( "[" "]" )*         // NewArray

BinOp::= ("*"|"/"|"%"|"+"|...)                                  // BinOp
PreOp::= ("+"|"-"|"~"|"!"|"++"|"--")                            // PreOp
PostOp::= ("++"|"--")                                           // PostOp
AssignmentOp::= ("="|"+="|"-="|...)                             // AssignmentOp
```

Familiarize yourself with the `AST.java` and all its sub-classes. I cannot stress this enough. The more you know about the nodes, the easier the subsequent phases will be.

2.4 Typing the Non-terminals

As you may recall, all the non-terminals were typed as *AST* in phase 1. This would in theory still work as all the parse tree nodes are sub-classes of AST, and you may assign a subclass instance to a super-class variable. However, you should (read MUST) go through the list of non-terminals and provide the appropriate type, i.e., the class name of the parse tree node generated in the actions associated with the rule that has the non-terminal in question on its left-hand-side. This information is changed at the top of the CUP file.

2.5 Work List

- Determine the types of all the non-terminals. Some are pretty obvious, like for example the if statement. Naturally the non-terminals *if_then_statement*, *if_then_else_statement*, and *if_then_else_statement_no_short_if* are all of type `IfStat` (i.e., they create an object of the `IfStat` class). Once you have determine this you can use the types in the constructor of the `IfStat` to determine the types of the individual parts of the if statement. Start with the obvious non-terminals and work your way back like I just explained.

- Add action to your *cup* file. That is, add {: `RESULT = ...` :} to each production rule, where `...` creates an object of the right type. This will always be one of the classes in the parse tree node files, or `null`. Keep in mind that if one right-hand-side of a production creates a sequence, then all right-hand-sides should create a sequence. For example, if we have a production like

  ```
  A ::= A a
     | a
  ```

  we are dealing with a repetition or a list of `a`s. The typical action you would add looks like this:

```
A ::= A:lst a:e          {:  RESULT = lst.append(e); :}
   | a:e                  {:  RESULT = new Sequence(e); :}
```

As you can see, both right-hand-sides return a sequence, so the type of the entire production is *Sequence*. Now if we have the following production in addition to the above one:

```
A_opt ::= A
        | ϵ
```

then the first right-hand side will return a *Sequence*, so in order to type this correctly, the empty production must also return a *Sequence*. We solve this problem in the following way:

```
A_opt ::= A:lst          {:  RESULT = lst; :}
        | ϵ               {:  RESULT = new Sequence(); :}
```

of course you only need to use this technique if the non-empty right-hand-side returns a sequence.

- Compile your compiler using the `ant` command.
- Run the `espressoc` script to test your compiler.
- Compare your results to my reference implementation.

2.6 Espresso Imports

Espresso supports imports; not imports in the same way as Java, because Java supports packages, Espresso does not. In Espresso the import statement appears at the beginning of the source file, and takes the name of a file to import. The file being imported should be placed in the `Include` directory. See section B for more on how to write libraries. You do not have to do anything in this phase to deal with imports; it is already done for you!

2.7 Obtaining the Handout Code

You can use the `espressou` script to install phase 2 like you did phase 1: `espressou install 2`.

2.8 Running the Reference Implementation

In order to check your compiler against my version of the **Espresso** compiler, you can use the `espressocr` script. That will submit your file to my reference implementation and return the output.

2.9 Pre-supplied Code and Testing

When you hand in your project we will use the `build.xml` file and the `ant` command to make your project, and the `espressoc` script to run your compiler. If we cannot do that we will not mark your project, i.e., you get 0 on this phase. We will do the following to test your compiler:

```
ant clean
ant
espressoc <testfile.java>
```

so please make sure this process works on `java.cs.unlv.edu`.

2.10 Due Date

The second phase of the project is due ____________________________ before midnight.

Chapter 3

Phase 3—Symbol Table—Definition and Resolution

3.1 Introduction

Phase 3 of the **Espresso** project deals with defining and resolving symbols, or names of variables, classes and methods. The phase consists of two passes through the parse tree:

1. A definition pass where classes, interfaces, methods and fields are inserted into a symbol table (also sometimes called an identification table). We need to perform this pass before resolving names because classes, fields and methods can be referenced before they are declared.

2. A resolution pass where method parameters and local variables are inserted into the symbol table, and uses of variables, parameters, methods, classes etc. are looked up in the table.

The symbol table phase will make use of a technique for traversing the parse tree that we will use over and over again. You will be using a visitor pattern for traversing the parse tree. More about this pattern later on. You will be doing all your work in 2 files: `ClassAndMemberFinder.java` and `NameChecker.java`.

3.2 Scoping

This third phase is concerned with name usage checking, which means checking that the scoping rules of the language have not been violated (i.e, no names are referenced in places where they cannot be resolved to match a declaration/binding of that name).

Espresso, like Java, has static scoping. The nice thing about static scoping is that all name-use checks can be performed at compile time, or in other words, we can check that all reference to classes, fields, methods, parameters, and local variables are legal at compile time.

We define the *scope* of a name (bound by a declaration of a class, method, field, parameter, or local variable) as the place in the code where that name can be legally referenced (and where the use resolves to the declaration/definition in mind!)

Let us look at the five different declarations that are possible in **Espresso**:

- **Classes**: The scope of a class declaration is everywhere. That is, a class name can be legally referenced anywhere in the code, even before it has been declared (This is why we need 2 passes through the parse tree in this phase). Classes cannot be re-implemented, that is, only one class of each name is allowed.

- **Fields**: The scope of a field is the class in which it was declared (i.e., the body of methods, constructors and static initializers as well as in initializers of other fields that are declared after the field in question). In addition, the scope of a field is extended to any sub-classes of the class in which the field was declared, but **only** if it was not declared `private`. Fields cannot be overloaded/re-implemented in **Espresso** (they can in Java, which causes all sorts of trouble; look in appendix A for a little example to test your Java skills)

- **Methods**: The scope of a method is just like that of a field: the entire class in which is was declared as well as any subclass (unless it was declared `private`). In addition method can legally be referenced by *obj.method*() or *class.method*(). Method **can** be overloaded, but must all have the same return type. More about method overloading later in this chapter.

- **Local variables & parameters**: The scope of a (formal) parameter is the body of the method in which it is a parameter. A local variables' scope is the rest of the block (starting immediately after the declaration of the variable) in which the declaration appears. Locals can hide parameters, but not other locals in the same block (Though in **Espresso** a *for*-loop opens a new scope which allows you to hide locals and parameters—Java handles this in a slightly different manner). Here are a few example of some legal and one illegal code pieces.

void f(**int** a) { **int** *a*; . . . }	**void** f(**int** a) { **int** *a*; **int** *a*; . . . }	**void** f(**int** a) { **int** *a*; **if** (. . .) { **int** *a*; . . . } }	**void** f(**int** a) { **int** *a*; **for**(**int** *a*;. . .) { . . . } }
Legal	Illegal	Legal	Legal

Understanding these scoping rules is important, but it does not really help us implement a pass though the parse tree that checks that all names used are in scope. To do that, we have to specify how name resolution progresses. Most of the time, we know what type of a name (i.e., class, method, etc.) we are looking for, but not always.

We keep classes in a global class table. Each class has its own method field table, and each method a table with parameters and each block a table with its locals.

The fist three (classes, fields, and methods) can be referenced before they are defined, so they must be put into the appropriate tables **before** the name resolution phase.

Parameters and locals are always looked up in the table associated with the most current block, and each time a new block is encountered, and a new table created, it is chained to the previous one. That way we can look in the "closest enclosing scope" for a local or a parameter, and if not found repeat the process with the next closest enclosing scope etc.

3.3 Visitor Pattern

In this phase we are going to use the *visitor* pattern for the first time—in subsequent phases we will reuse the visitor pattern frequently. The visitor pattern consists of 2 parts. A general *Visitor* class, and an implementation of a method *visit* for each parse tree node that is to be visited. The visitor pattern uses a double dispatch method to visit all (or some) nodes in the parse tree. The *Visitor* class can be found in the file `Visitor.java`, and it looks like this:

```
public abstract class Visitor {
  public Object visitAssignment(Assignment as) {
    return as.visitChildren(this);
  }
  public Object visitBinaryExpr(BinaryExpr be) {
    return be.visitChildren(this);
  }
  .
  .
}
```

The *Visitor*[1] pattern contains one method *visitXXX()* for each parse tree node *XXX*.

Here is an example of how the *visit* method looks for the binary expression parse tree node:

```
  public Object visit(Visitor v) {
    return v.visitBinaryExpr(this);
  }
```

Each of these methods (*visitBinaryExpr()* in this case) will call the *visitchildren()* in the *AST* class, that in turn will invoke the visit method on all its children passing the visitor pattern as the argument. You will see that if no *visitXXX()* method has been specified in the re-implementation of the visitor pattern, then the original one in `Visitor.java` will be called. Thus passing the visitor recursively to all its children. The visitor assures that all nodes are visited, and the specific implementation of the pattern allows you to control which nodes you want to visit during the traversal of the tree. Looking at the *visit()* method in the parse tree node you will see that all it does is call back into the visitor pattern object to the method named *visitXXX()* where *XXX* is the name of the parse tree node.

Now, to implement a traversal of the parse tree you must implement the *Visitor* pattern. This means that you must re-implement all the *visitXXX()* methods for the parse tree nodes that **you** want to visit during the traversal. If you do not re-implement a method *visitXXX()* the standard implementation (which just calls on all the children, thus skipping the node itself) will be executed when called from the parse tree node.

Example: If we wanted to write a traversal of the parse tree that did something for only the *BinaryExpr* parse tree nodes we could do it in the following way:

```
  public class TestVisitor extends Visitor {

    public Object visitBinaryExpr(BinaryExpr be) {
      // What ever you wish to do with the binary expression goes here.
      // Now go visit the children:
      be.visitchildren(this);
    }
  }
```

Now you can create a new instance of this visitor class and pass it to the *visit* method of the root of the parse tree:

```
  root.visit(new TestVisitor());
```

As mentioned, we will need 2 passes of the parse tree in this phase: one to define classes, interfaces, methods, constructors and fields, and one to resolve all uses of names and associate uses of names with their corresponding declarations by setting the *myDecl* field.

As an example of a visitor implementation you might want to take a look at the `PrintVisitor.java`, which I use to traverse the tree for pretty printing purposes. This file should be in your hand

[1]To learn more about the visitor pattern you can go to
`http://www.javaworld.com/javaworld/javatips/jw-javatip98.html`

out directory.

Hint: If you find that your traversal of the parse tree is cut short, or in other words, there are parts of the tree that does not get visited, it is probably because you forgot to visit the children of a node; that is, you forgot a call to `visitChildren(...);`, or alternatively `super.visitXXX(...);`

3.4 Methods in Espresso

Often the word *method* is used in object oriented languages to denote a procedure or a function. Strictly speaking, a procedure does not return a value, but a function does. Other languages have other, stricter, rules that define what a procedure and a function is, but in Espresso we refer to all such entities as *methods*.

A method in Espresso is like a method in Java. A method can take a number of parameters (with call-by-value semantic) and return max one value. To be able to tell methods apart in later phases we need to introduce a *method signature*. A method signature is made up of three different parts: The name of the method, the signature of the parameters and the signature of the return type. See the table later in this chapter for signatures of atomic types and classes as well as arrays.
Espresso, like Java, has two special kinds of methods: constructors and static initializers. A constructor is a method that is named the same as the class in which it appears, and it has no return type. A constructor is called automatically when an object is constructed. The parameters passed to *new* determines which constructor is executed. If no constructor is present for a class, the compiler will automatically generate one. This constructor is referred to as the *default constructor*; it takes no parameters and, if auto generated, does nothing but invoke the default constructor in the super class (This is not entirely true, if there are non-static fields that need to be initialized, then this code will happen here as well). The internal name of a constructor, rather than using the class name, is `<init>`.

The second special method is the static initializer; a static initializer is to a class what constructors are to objects. Remember, a constructor (or initializer, as it is sometimes called) is executed when the object is created. That is, it is executed **once** for each object. A static initializer or a class initializer is executed **once for each class**, typically when the class is loaded by the virtual machine. A static initializer can be specified explicitly in a class by `static` { `...` }, and if none is specified but one is needed (e.g., to initialize static fields), one will be automatically generated. A static initializer is named `<clinit>`.

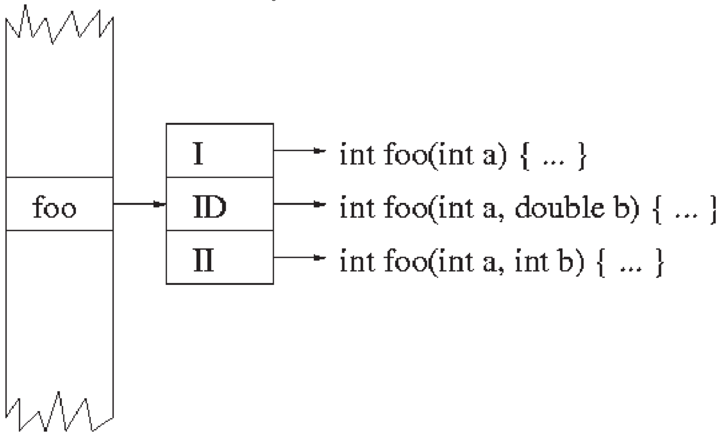
3.5 Method Overloading

Espresso allows method overloading. This means that the same method name can be used many times within the same class (or in the super class) as long as the parameters signatures are different. The following example is valid **Espresso** code:

```
public class A {
  int foo(int a, int b) {
    ...
  }
}

public class B extends A {
  int foo(int a) {
    ...
  }
  int foo(int a, double b) {
    ...
  }
  int foo(int a, int b) {
    ...
  }
}
```

This means that it is not enough to just store the name of a method in the symbol table as there could be more than one method with the same name. The solution to this problem will be resolved as follows: The key to the method table is the name of the method, and the value stored is another symbol table where the key is the (parameter) signature of the methods, that means the `B` class above would have a symbol table that looks like this:


foo
I
ID
II
int foo(int a) { ... }
int foo(int a, double b) { ... }
int foo(int a, int b) { ... }

so to insert a method into the method table (each class declaration has its own table for methods and one for fields as well) we would need to look for it first—and then insert it into this table. If no such table exists, we would have to make one and insert the method in to it using the method's parameter signature as a key, and then insert the entire table into the method table using the method name as a key.

In the name resolution phase, when looking for a method we will unfortunately not be able to determine if a method with the correct signature exists—to do that we need to compute the types of the actual parameters for each invocation, something we will not do until phase 4. Instead we will simply determine if there is any method of the right name in the class hierarchy of the invocation of the method. To do this you need to implement a method that searches recursively through the class hierarchy of a class. The search for a method should follow the following model:

1. Look in the class of the invocation in question.

2. If nothing is found there repeat from step 1 using the super class if such a class exists.

3. If still noting was found repeat step 1 with all the interfaces implemented by the current class. (Do this step only if you implement **Espresso+**)

If no method of the right name was found an error message must be displayed. This resolution can only be done if the target expression of the invocation is simple enough to determine the target class, that is, the class in which the method table that should contain the method in question.

3.6 Symbol Table Structure and Scope

A symbol table is a hash table that links symbols to their definition point, i.e., variables to their declarations (*LocalVarDecl, ParamDecl* or *FieldDecl*), classes and interfaces to their *ClassDecl* and methods to their *MethodDecls*, and constructors to their *ConstructorDecl* and static initializer to their *StaticInitializerDecl*. We use a number of different symbol tables in this phase:

- The *Compilation* parse tree node has a symbol table that holds all classes and interfaces in the program. The key of this table is the name of the class or interface.

- The *ClassDecl* parse tree node has 2 symbol tables: one for fields and one for methods. The method table was described in detail in the previous section (the key at the top level is the method name, and at the method level the key is the parameter signature). The field table is indexed by the name of the field.

These tables are always there. The following tables are allocated when new scopes are opened. Keep in mind that the class table found in the *Compilation* node is the top scope, the ones found in the *ClassDecl* nodes are the next ones in the scope hierarchy. Following these are (in no significant order):

- The new scope created by a *Block* (containing local variables). A block is a set of curly brackets: { ... }. and any local variable declared inside a block can only be referenced in the remainder of the block, that is, until the closing }.

- The new scope created by a *ForStat* (possibly containing any newly introduced loop variables). Since a variable can be declared in a for construct, the scope of such a variable is the rest of the for statement as well as the body.

- The new scope created by a *MethodDecl* (containing the parameters of the method).

- The new scope created by a *ConstructorDecl* (containing the parameters of the constructor).

Since a method and a constructor's body is a block, the parameters and the locals are entered into different symbol tables, thus it is legal to have parameters and locals that are named the same; however, the local variable will overshadow the parameter, which can never be referenced.

We will keep a global variable called *currentScope* within the *NameChecker* that always holds a reference to the the current scope. The current scope is the more recently opened scope.

When we encounter a *ClassDecl*, we make the class' field table the top most symbol table. That way we can easily find fields which are referenced later without a target (i.e., fields that are references simply by their name without an object or a class and a dot). Any symbol table knows its parent, so when resolving names there is a static scope chain of symbol tables linked that we can follow all the way to the top-most class table.

Recall that static scoping means that all uses of variables, method, fields, classes, and parameters can be resolved at compile time.

Study the `SymbolTable.java` file and keep in mind that a new scope can be opened with the method *newScope()* and the current scope can be closed with *closeScope()*.

3.7 Definition Pass

As mentioned, this phase requires two passes through the parse tree. This means you will need two different visitor implementations. The first one we call *ClassAndMemberFinder*, and the skeleton code can be found in the `ClassAndMemberFinder.java` file.

The main purpose of this pass through the parse tree is to define classes, methods and fields. The reason we need to define these parts before we go into name resolution is, that classes, fields and methods can all be referenced before they are declared (forward referenced), i.e., you can call a method that is textually further down in the code, so we need to insert all classes into the global class table and all the methods and fields of each class into their respective method or field table in the class.

I have implemented *addMethod()* and *addField()* to add methods/constructors and fields to the symbol tables in the *ClassDecl* parse tree node. Take a look at them (they are in the `ClassAndMemberFinder.java` file) and make sure you understand what is going on!

You will need to re-implement a number of the *visitXXX()* methods, namely for all the parse tree nodes that deal with declaration of classes, methods, constructors, fields, and static initializers. Keep in mind that constructors should be inserted into the method table with the special name `<init>`, and static initializers are named `<clinit>`.

3.8 Resolution Pass

The resolution phase is a traversal of the parse tree where every use of a variable, class, method, field, constructor, or parameter is checked, i.e., looked up in the symbol tables, and where every definition of a local or a parameter is inserted into the current scope. Furthermore, all uses will be associated with their definitions (i.e., each use of a variable will be associated with where it was defined etc.) by setting the variable *myDecl* found in many parse tree nodes—this makes it much easier to find the definition in the future, thus avoiding a look up in the symbol tables.

Recall that Espresso has static scoping, which means that we can statically check all name uses. Classes and interfaces are already in the global class table, and fields and methods are already in each class' field table/method table. This leaves only local and parameters to be correctly inserted into a symbol table structure.

Each language construct that represents a scope should open a new scope (i.e., create a new symbol table and make it the current scope (*currentScope* is a global variable holding a reference to the symbol table that represents the current scope). When a new scope is opened its parent scope will be the symbol table that was the current scope at the time it was opened. This creates a chain of symbol tables representing the static scoping rules of Espresso. A new scope can be opened in the following way:

```
currentScope = currentScope.newScope();
```

and when a scope is closed you can remove the corresponding symbol table and restore the parent symbol table as the current scope:

```
currentScope = currentScope.closeScope();
```

All definitions (of locals and parameters) should be inserted into the current scope and all uses of classes should be looked up in the global class table, and all uses of methods should be looked up in the method table found in the method table inside the respective class. Fields represented by a *fieldRef* (i.e., fields with a target) can only be resolved if the target is *this* (technically we could do it for a *super* target, but it is more easily left for later. We can do it if the target is *null*, but then we would not have an instance of a $FieldRef$ but rather a $NameExpr$. When visiting a *ClassDecl* you should update the global *currentClass* so we can always get to the current class and its tables.

NameExpr

It is vital that you feel comfortable with the *NameExpr* parse tree node as you will need to work with it a lot in the future phases. In general, an instance of the *NameExpr* node represents the **use** of a variable as an expression. For example, `a + b`, both `a` and `b` will be represented by instances of *NameExpr*. However, an instance of a *NameExpr* can also represent a field without a target. After this phase, we can rewrite the parse tree based on *NameExpr*'s *myDecl*s. If it is a field reference, we replace it by a *FieldRef*. See the section on rewriting later on in this chapter. It is impossible to determine if `a` is a parameter, local variable of a field in the class without looking at its corresponding declaration. In some cases a *NameExpr* can also represent the name of a class.

By making the field table of the current class the top symbol table in the chain of tables, all uses can be resolved by the same lookup; namely a lookup in the current scope. Fields will be found in the very last symbol table in this situation, but the lookup mechanism automatically searches in the parent tables if nothing is found in the current table. Locals and parameters are inserted into the current scope when they are declared and resolved (i.e., looked up in the current scope) when they are used. All uses of parameters and locals are represented by a *NameExpr*. So, a *NameExpr* should be looked up in the scope chain or the class table.

The resolution pass should be implemented in the `NameChecker.java` file. Apart from another instantiation of the *Visitor* pattern, you will need to implement a number of helper methods described below.

myDecl

When resolving the use of a name we check that the variable referenced is in scope. That is, we check that it is legal to refer to the variable at this place in the code. In subsequent phases we will need to consider the declaration associated with the use of a variable. For example, in phase 4 we need to access the type of a variable in order to perform type checking. We could of course look the variable up in the symbol table structure every time we need to find its declaration, but instead of doing that (and thus having to keep all the symbol table around for the subsequent phases), we can associate all uses with their declarations. That can easily be done by having a field in all parse tree nodes that subclass *Expression*, named *myDecl*, point to the declaration: Once the variable name has been looked up in the symbol table structure and its declaration has been resolved, we set the node's *myDecl* field to point to this declaration; in later phases we can then simply refer to this field when we need the declaration associated with the use.

3.8.1 *getMethod()* and *getField()*

You will need to implement the 2 methods *getMethod()* and *getField()*. I have left their implementation blank so you must fill in the missing pieces.

getMethod() traverses the class hierarchy to look for a method of name 'methodName'. We return the *SymbolTable* with all the methods of the same name, if we find any method with the correct name, and *null* if we do not. Since we do not have types yet, we cannot look at the signature of the method to determine if the signature of the invocations parameters matches, so all we do for now is look if any method is defined. The search progresses as follows:

1. Look in the current class.
2. Look in its super class, if there is one.
3. Look in the interfaces that the class implements, if there are any.

Remember that the an entry in the *methodTable* is a symbol table it self. It holds all entries of the same name, but with different signatures. The *getField()* searches in the exact same way.

3.8.2 *getClassHierarchyMethods()*

Next, you will need to implement a method that takes in a class and an empty sequence and builds a sequence of all the methods found in the entire class hierarchy, that means, in the class, its super class, its interfaces, the super classes super class, and interfaces, the interfaces interfaces etc.

3.8.3 *checkReturnTypesOfIdenticalMethods()*

Now you have collected a sequence of all the methods in a class hierarchy. Since **Espresso** allows method overloading and inheritance we need to check that if a method has been re-implemented is has been re-implemented with the same return type. The following example is **not** valid **Espresso** code:

```
public class A {
  int foo(int a) { ...}
}

public class B extends A {
  void foo(int a) { ...}
}
```

The *foo()* method with parameter signature I (more about signatures shortly) has return type *int* in class *A* and return type *void* in class *B*, and *A* is *B*'s super class. Each type has a 'signature' the following table summarizes the signatures for the different **Espresso** types:

Type Name	Type Signature
boolean	`Z`
byte	`B`
short	`S`
char	`C`
int	`I`
long	`L`
float	`F`
double	`D`
String	`Ljava/lang/String;`
void	`V`
class type 'myClass'	`LmyClass;`
array of `T`	`[T`

Where 'myClass' is an example of any class name. For example if we have the following method:

```
int foo(myClass m, String str, int a, int b, boolean b) {
  ...
}
```

the parameter signature would be the string `"LmyClass;Ljava/lang/String;IIZ"`

Any methods parameter signature can be obtained by a call to its *paramSignature()* method, and any types signature by a call to its *signature()* method.

You must write some code (*checkReturnTypesOfIdenticalMethods()*) that for each method (not constructors!) traverses the class' hierarchy and compares the return type with the return type of any other method with

- The same name.
- The same parameter signature.

3.8.4 *checkUniqueFields()*

Since **Espresso** is a subset of Java we can be as strict or relaxed as we want (or as I want ;-)), so I have decided that we do not allow fields to be 're-implemented' in sub classes. The *checkUniqueFields()* method will insert all its fields into the fields *Sequence* and call recursively on all its super classes and interfaces, who in turn will traverse the field *Sequence* with each one of its fields to check if it is implemented in a sub class or a class that implement an interface. If this is the case an error should be signaled.

3.8.5 NameChecker

Now that we have written helper methods (which by the way all will be called from *visitClassDecl()*) we can start on the pass through the parse tree, i.e., the implementation of a name checker visitor.

In this pass you must re-implement *visitXXX()* for all parse tree nodes that either create a new scope or has a name of a variable, method, field, class etc. that needs to be resolved, i.e., looked up in the symbol table structure. Once you have found the definition of what you were looking for you should set the corresponding *myDecl* field in the parse tree node.

Do not worry about names like *xxx.yyy*. Without type checking we cannot check that *yyy* is a valid name. only if *xxx* is '*this*' can we handle it now. The same goes for invocations: only if the target (i.e., *xxx* is *null* or '*this*' can we check something like *xxx.yyy(...)*.

In my reference implementation I have re-implemented 13 *visitXXX()* methods.

With these 2 visitors and their helper methods implemented you should have completed phase 3. The `Driver.java` driver file is already set up to call both passes through the parse tree, so you should test it against my implementation. Remember that for this phase it is as important to test your implementation on a bad program as it is on a good program.

3.9 Espresso+

The biggest job for this phase if you wish to implement **Espresso+** will be the following two points:

- Checking everything that has to do with the modifier abstract.
- Checking that classes that implement interfaces, and inherit from abstract super classes implement all the abstract bodiless methods.

3.9.1 *getClassHierarchyMethods()*

As mentioned you will need to do some further checking in this method. The following checks applies to methods:

- If a method does not have a body, and appears in a class, then the method should be declared abstract.
- If a method does not have a body, and appears in a class, then the class should be declared abstract.

If a class is declared abstract it cannot have any constructors. We do not need to worry about this problem with interfaces because the grammar simply does not allow to have methods without a return type in an interface.

3.9.2 *checkImplementationOfAbstractClasses()*

If we have abstract classes and interfaces we need to assure that all classes that are not abstract do not end up without an implementation of an abstract or an interface method. We do this (or one way of doing it) by separating the sequence of methods that we build in the *getClassHierarchyMethods()* method into a sequence that contains all the abstract and all the interface methods, and on that contains all the regular methods with bodies. Then we need to make sure that every single method that is either abstract or an interface method has an implementation in the sequence of regular methods. We need to compare name, parameter signature and return type signature to assure that the implementation is valid. If there is a method that does not have an implementation an error must be signaled.

3.10 Common superclass 'Object' in Espresso

Like in Java, Espresso benefits greatly by having a common super class for all classes that do not declare a super class explicitly. In *visitClassDecl* we explicitly add *Object* as the super class if one does not already exist. This of course means that a class by the name *Object* must be available. Such a class is explicitly inserted into the tree in the file *Phases/Phase2.java*; it contains nothing but a default constructor (which when we generate code will contain an explicit constructor invocation to the default constructor in *java/lang/Object*, as this is a necessity for our code to run correctly on the Java Virtual Machine. You should not have to do anything in your coding for all this to work.

3.11 Parse Tree Rewriting

In the file `Rewrite.java` in the `Utilities` directory we perform a parse tree rewrite. You do not have to do anything here, but you should be aware of what is happening. Recall, sometimes a *NameExpr* represents a field reference without a target. In reality, this should be a *FieldRef* with a target of *null*. This rewriting visitor replaces the *NameExpr* that represent field references with *FieldRef*s where the target is either *This* if the field is non-static, or the name of the class if the field is static.

3.12 Work List

- **Read and understand** the Visitor Pattern. This is very important before you start implementing.
- Read and understand section 3.3 and 3.4. It is important that you understand what *scope* is and how it works in Espresso.
- Implement the definition pass in the `ClassAndMemberFinder.java` file.
- Implement the resolution pass in the `NameChecker.java` file. Make sure you understand the idea behind the use of the field *MyDecl*. It is used in the following way: Every time we have a use of something we have to look it up in the table structure. If the use is legal the corresponding declaration will be returned from the table look-up. This declaration is what goes into the *myDecl* field. This way we do not have to look it up next time we need the declaration associated with the use of something; also this means that once this phase is done we can discard of most of the symbol tables (specifically, all the symbol tables created in the resolution pass.)
- Use the *build.xml* and `ant` to compile your compiler.
- Use the 'espressoc' script to test your compiler.

- If you wish to compare my output you can use the 'espressocr' script to invoke my reference compiler. Most people find it advantageous to mimic the output of my compiler such that they can use the UNIX utility 'diff' to compare the output from the two compilers.

3.13 Obtaining the Handout Code

Use `espressou install 3` to install the phase 3 handout code.

3.14 Running the Reference Implementation

In order to check your compiler against my version of the **Espresso** compiler, you can use the `espressocr` script. That will submit your file to my reference implementation and return the output.

3.15 Pre-supplied Code and Testing

The pre-supplied code for this phase is my solution to phase 2 (i.e., *CUP* and *JFlex* files that work) plus all the parse tree nodes that you already have.

You will find the `SymbolTable.java` file, the skeletons for `ClassAndMemberFinder.java` and `NameChecker.java` and a `Type.java` that is used to generate signatures. `Visitor.java` and `PrintVisitor.java` are also supplied. When you hand in your project we will use the `build.xml` file to compile your project, and the `espressoc` script to run your compiler. If we cannot do that we will not mark your project, i.e., you get 0 on this phase. We will do the following to test your compiler:

```
ant clean
ant
espressoc <testfile.java>
```

so please make sure this process works on `java.cs.unlv.edu`.

3.16 Due Date

The third phase of the project is due ____________________________ before midnight.

Chapter 4

Phase 4—Type Checking

4.1 Introduction

Phase 4 of the **Espresso**/**Espresso+** project deals with type checking. Type checking is, as you know, the task of assuring that a program is sound with respect to its use and definition of types. A simple example is the following binary expression

```
a = 89 + "Hello";
```

which obviously does not type check; adding a string to a number is illegal. A more interesting example is

```
int a;
a = 89 + 1.2;
```

This looks OK, but looking closer at it and doing a little analysis we get the following:

Type(a) = Int
Type(89) = Int
Type(1.2) = Double
Int + Float = Double, so
Type(89 + 1.2) = Double,

which is not assignable to an Int.

Apart from checking types in this phase we also need to deal with field references and invocations that are of the form *target.x* or *target.f()* where *target* is not *null* or *this* or *super*. An example could be *x.f.g().f*. This becomes much easier when we deal with types because classes are types as well, and they contain field and method tables to perform the look up in. In this situation, the type of *x.f.g()* must be an object and contain a field named *f*, that is, *g()* must return an object which has a field named *f*. *x.f* must be an object containing a method *g()*, and finally *x* must be an object with a field *f* which holds an object reference to an object with a method named *g*.

When implementing your type checker you need to re-implement *visitXXX()* for each node in the parse tree that either needs to be type checked and/or represents/has a type.

The body of the *visitXXX()* function will do the type checking, and if there is no checking to be done because the node represents a type/has a type, this type is simply returned. Here is an example from my solution:

```
/** CLASSTYPE */
public Object visitClassType(ClassType ct) {
  println(ct.line + ": Visiting a class type");

  return ct;
}
```

A *ClassType* does not require any checking, but it is in itself a type, so simply return this type. All parse tree nodes that are descendants of *Expression* must return their type after the type checking has taken place (i.e., in the example above, after checking that 89 and 1.2 are types compatible with the + operator, the result type of the + operation between an integer and a double is returned; double in this case).

Also remember, that if you are not at a leaf node in the parse tree then you must make the type checker continue down into the tree; you must either call *visit()* explicitly on the children that must be type checked or you can call *super.visitXXX()* to type check what is below the node in the tree.

So a rule of thumb: calling *visitXXX()* will type check the node and its children, and return the type of the expression/statement if it has a type (if not, it will return *null*). This means that a type for something can be obtained by calling *visit()* on the object representing it.

Also note, any parse tree node that has a type (typically something that extends *Expression* contains a field named *type* of type *Type*. Once you have computed the type of the node, you should set this field before returning it.

As with phase 3, we will be using an instance of the *Visitor* pattern to do the traversal of the parse tree, in this case it is called *TypeChecker*, and can be found in the `TypeChecker.java` file.

You should familiarize yourself with the methods in `Type.java`, especially the *identical()*, *isSuper()* and *assignmentCompatible()*.

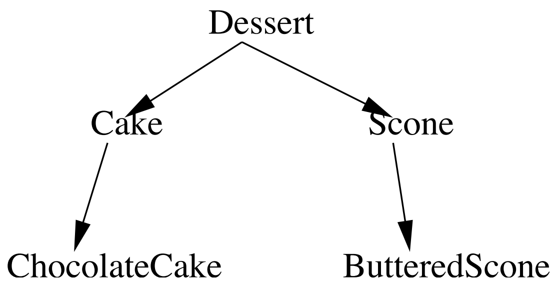
Also remember, chapter 7 contains more information about type checking for arrays.

4.2 Finding the Right Method to Call

Since we have function overloading in **Espresso**, determining which method to call is not always easy. The following is a copy of pages 108-111 of "The Java Programming Language" by Ken Arnold and James Gosling:

> Members of objects are accessed through the . operator, as in *obj.method()*. You can also use the . to access static members, using either a class name or an object reference.
>
> For an invocation of a method to be correct, arguments of the proper number and type must be provided so that exactly one matching method can be found in the class. If a method is not overloaded, this is simple, because only one parameter count is associated with the method name. Matching is also simple if there is only one method declared with name and number of arguments provided. If two or more overloads of a method have the same number of parameters, choosing the correct method overload is more complex. Java (and **Espresso**!) uses a "most specific" algorithm to do the match:
>
> Find all the methods that could possibly apply to the invocation, namely all the overloaded methods that have the correct name and whose parameters are of types that can be assigned the values of all the arguments. If one method matches exactly for all arguments, you invoke that method. If any method in the set has parameter types that are all assignable to any other method in the set, the other method is removed from the set because it is less specific. Repeat until no elimination can be made. If you are left with one method, that method is the most specific and will be

the one invoked. If you have more than one method left, then the call is ambiguous, there being no most specific method, and the invoking code is invalid. For instance, suppose you have an the following class hierarchy:



Further suppose you had several overloaded methods that took particular combinations of *Dessert* parameters:

```
void moorge(Dessert d, Scone s)        { /* 1st form */ }
void moorge(Cake c, Dessert d)         { /* 2nd form */ }
void moorge(ChocolateCake cc, Scone s) { /* 3rd form */ }
```

Now, consider the following invocations of *moorge()*:

```
moorge(dessertRef, sconeRef);
moorge(chocolateCakeRef, dessertRef);
moorge(chocolateCakeRef, butteredSconeRef);
moorge(cakeRef, sconeRef);                // INVALID
```

The first invocation uses the first form of *moorge()*, because the parameters and argument types match exactly. The second invocation uses the second form, because it is the only form for which the provided arguments can be assigned to the parameter types. In both these cases, the method to invoke is clear after step 1 in the algorithm described above.

The third invocation required more thought. The list of potential overloads includes all three forms, because a *chocolateCakeRef* is assignable to any of the first parameter types, a *butteredScone* reference is assignable to either of the second parameter types, and none of the signatures matches exactly. So after step 1 you have a set of three candidate methods.

Step 2 requires you to eliminate less specific methods from the set. In this case, the first form is removed from the set because the third form is more specific; a *chocolateCake* reference can be assigned to the first form's *Dessert* parameter and a *Scone* reference can be assigned to the first form's *Scone* parameter, so the first form is less specific. The second form is removed from the set in a similar manner. After this, the set of possible methods has been reduced to one-the third form of *moorge()*—and that method will be invoked.

The final invocation is invalid. After step 1, the set of possible matches includes the first and second form. Since neither form's parameters are assignable to the other, neither can be removed form the set in step 2. Therefore, you have an ambiguous call that cannot be resolved by the compiler, and therefore an invalid invocation of *moorge()*.

These rules apply to the primitive types too. An *int*, for example, can be assigned to a *float*, and resolving an overloaded invocation will take that into account just the way it considered that *ButteredScone* reference was assignable to a *Scone* reference.

Methods may not differ only in return type. If, for example, there were two *doppelganger* methods that differed only because one returned an int and the other a short, both would make equal sense in the following statement:

```
double d = doppelganger();
```

You must implement this algorithm in the method *findMethod()* in `TypeChecker.java`. *findMethod()* takes in a sequence of concrete methods (you computed this sequence in the previous phase, and I added them to the *ClassDecl* node), the name of the method we are looking to invoke, and a sequence of the actual parameters. Remember, since the actual parameters are expressions, and the formal ones are declarations you can call *type()* on both of them. Also remember that the methods you are working with in *findMethod()* can be both constructors and methods. Here is the example described above. Your compiler should be able to pick the correct method to invoke for the first 3 invocations, and it should complain about the last one.

You can use the *listCandidates()* method found in `TypeChecker.java` to print out the possible methods if your compiler determines that an invocation is ambiguous.

```
public class Dessert {}
public class Cake extends Dessert {}
public class Scone extends Dessert {}
public class ChocolateCake extends Cake {}
public class ButteredScone extends Scone {}

public class test1 {
    void moorge(Dessert d, Scone s) {}
    void moorge(Cake c, Dessert d) {}
    void moorge(ChocolateCake cc, Scone s) {}
    void main() {
        Dessert dessertRef;
        Cake cakeRef;
        Scone sconeRef;
        ChocolateCake chocolateCakeRef;
        ButteredScone butteredSconeRef;

        moorge(dessertRef, sconeRef);
        moorge(chocolateCakeRef, dessertRef);
        moorge(chocolateCakeRef, butteredSconeRef);
        moorge(cakeRef, sconeRef);
    }
}
```

4.3 Espresso+ and Espresso*

The type checking phase for **Espresso+** is rather simple once you have done the checking with respect to abstract methods. The only extra thing that needs to be type checked for **Espresso+** is the `switch` statement and the ternary expression (`<test> ? <expr> : <expr>`), you should have no problem finishing type checking for **Espresso+**. All the extra information needed for type checking **Espresso*** (i.e., arrays) can be found in chapter 7.

4.4 Work List

- Familiarize yourself with the many methods in the file `Type.java`; many of these methods will do a lot of the checking for you.

- Determine which parse tree nodes have elements that need to be type checked. Each of these nodes will have a corresponding visit method in the type checker visitor.

- Determine, for each of these nodes, what needs to be type checked and what, if any, type the node represents (most of the nodes that return a type are expressions of some sort. For example, an if-statement does not return a type as it is a statement, but it has elements like the boolean expression that must be type checked. This boolean expression does have a type as it is an expression, and its type can be obtained by visiting it.)

- Determine which parse tree nodes represent types and implement a visit method for them to simply return their type when visited.

- Read and understand the section about how to determine which method to call and implement the *findMethod()* method.

- Use the `build.xml` file and the `ant` command to build your compiler.

- Use the `espressoc` script to test your compiler.

- Compare your compiler against my reference implementation (if you want.)

4.5 Test Files

You will find a directory of test files in `/home/CSC460/Espresso/Tests/Phase4/`. These test programs are the ones we will be using throughout the project. You are encouraged to make your own test programs in the same way, and place them in `/home/CSC460/Espresso/StudentTests/`.

We have created a large amount of test files for you for this phase—mostly bad test files, i.e., files that makes your compiler complain. In `/home/CSC460/Espresso/Tests/Phase4/BadTests/` we have provided you with one test file for each possible error message you can get by running the reference implementation. For example, to see which error messages you can get from a binary expression look for test files called `BinExprXX.java` where `XX` is a number. In other words, the test files will be named the same as the parse tree node they represent, and numbered if there is more than one (different) error message for each parse tree nodes.

Use this as a reference point for the things you need to check in your traversal trough the tree.

To determine whether your compiler is performing correctly, check it against the output from my compiler which you can execute using the `espressocr` script.

You can of course install the test files in your directory if you have not already done so. See appendix E for more information on how to do that!

4.6 Obtaining the Handout Code

Install the handout code with `espressou install 4`.

4.7 Running the Reference Implementation

In order to check your compiler against my version of the **Espresso** compiler, you can use the `espressocr` script. That will submit your file to my reference implementation and return the output.

4.8 Pre-supplied Code and Testing

The pre-supplied code for this phase is my solution to phase 3 plus the `TypeChecker.java` skeleton file. The file that you will be doing all your work in is `TypeChecker.java`.

When you hand in your project we will use the `build.xml` file to make your project, and the `espressoc` script to run your compiler. If we cannot do that we will not mark your project, i.e., you get 0 on this phase. We will do the following to test your compiler:

```
ant clean
ant
espressoc <testfile.java>
```

so please make sure this process works.

4.9 Due Date

The fourth phase of the project is due ____________________________ before midnight.

Chapter 5

Phase 5—Modifier Checking

Modifiers are a unique notion found in object oriented languages, and thus also in **Espresso**. They are used to control access to various parts of a class or an object. Java has a number of modifiers but **Espresso** only uses a couple of them. The complete list of **Espresso** modifiers is: `public`, `private`, `static`, `final`, and `abstract`.

Modifiers can be applied to classes, methods, fields, and constructors, but not all modifiers can be applied to every class/method/field/constructor. As stated, modifiers control access. For example, you cannot assign a value to a field declared `final` (except at declaration time), and you cannot inherit from a class that is `final`, and you cannot call a non `static` method from within a `static` method (static context), etc.

The following table shows which modifiers can be used where. To get an idea about what to check for, you can either refer to the list below, or you can look through the test programs provided.

	Class	Method	Field	Constructor
`private`	No	Yes	Yes	Yes
`public`	Yes	Yes	Yes	Yes
`static`	No	Yes	Yes	No
`final`	Yes	Yes	Yes	No
`abstract`[1]	Yes	Yes	No	No

Modifier list for **Espresso** components.

5.1 Static versus Non-Static

The checks associated with static/non-static are probably the hardest for this phase. In order to make this easier it is important that you fully understand what a static and a non-static context is.

You can think of a **non-static** context as a field or a piece of code residing in an **object**. That is, the value of a field found inside an object or the execution of a method or a constructor associated with an object. For example, any call of the form *object.method()* would be an execution of the method *method()* associated with object *object*. Consider the following code:

```
public class A {
    public static int i;
    public int j;

    public static void f() {
```

[1]`abstract` is for **Espresso+** only.

```
        i = 9;
        j = 10;
    }

    public void g() {
        i = 11;
        j = 12;
    }

    public static void h() {
        A a = new A();

        a.f();
        A.f();
        a.g();
        A.g();
    }
}
```

The code segment *A.i* is valid as *i* is a static field, that is, a class variable. Recall that class variables are associated with a class, and only one copy of them exist (as opposed to non-static fields, where each instance of the class gets its **own** copy of the field). Thus static fields can be accesses both through *object.field* and *class.field*, whereas non-static fields are associated with an object and can only be accessed through *object.field* (or just *field* if the contexts is a method or constructor inside the object containing the field). This means that *A.j* would **not** be legal as *j* is a non-static field.

The same argument holds for methods. Static methods can be called through *object.method()* and *class.method()* but non-static methods require a non-static context. If you consider the code above you can easily see why a non-static method requires a non-static context. If you consider method *g* running in an object *o*. method *g* accesses fields *i* and *j*. Field *i* is static, that is, it resides in the class and can thus be accessed; Field *j* resides in object *o* and can thus also be accessed. Now consider a call to *f*, e.g., *A.f()*. Again, field *i* can be accessed as it is stored in the class, but the reference to field *j* poses a problem here. Since *f* was called as a static method on a class the field *j* is not accessible; remember, it is associated with instances of the class, that is, objects, and can thus not be referenced from a method that is not executed on an object.

If we were to compile the code above with the Java compiler we would get the following errors (The **Espresso** compiler should produce the same errors):

```
(2) javac A.java
A.java:7: non-static variable j cannot be referenced from a static context
        j = 10;
        ^
A.java:21: non-static method g() cannot be referenced from a static context
        A.g();
         ^
2 errors
```

Now what about a call like *object.f()*? In this situation there is a context, that is, object *object* contains the field *j*. Should this be a legal call? The call looks legal; method *f* is declared static and called on an object, which seems perfectly legal; however, when the compiler checks method *f* it cannot know from where the method will be called, and since the method is declared static, when checking its body, only static members can be accessed, that is, the reference to *j* is illegal.

Convention dictates that if a method or a field is static, then it should be referenced through $< class - name > .field$ or $< class - name > .method()$, and if a method or a field is non-static though an object reference like $< object - ref > .field$ or $< object - ref > .method()$ (naturally there is no other way to do the last two whereas the first two would also work if $< class - name >$ were replaced with $< object - ref >$.)

5.1.1 The Current Context

In order to easily check many of the uses of fields and methods with respect to the modifier `static` and the use of methods and fields through *class.field* or *class.method()*, we need to know what the current context is; that is, we must be able to easily determine if the context is static or non-static. We do that by declaring a global variable called *currentContext*, which is of type *ClassBodyDecl*; recall, that we are always either in method, constructor, field declaration or a static initializer. A static initializer is always a static context, and the other three are static if their declaration is preceded by the `static` modifier. If you look in `ClassBodyDecl.java` you will find a method called *isStatic()* which determines if the context is static. You must remember to update this *currentContext* variable each time the context changes.

Also, the variable *leftHandSide*, which should be `true` when visiting the left hand side of an assignment. This makes it easy for you to check if you are assigning to a final field (which is illegal).

Things to Check

- References of non-static fields through a *FieldRef* like *class.field* is illegal.
- In a *MethodDecl*, a re-implementation of a static (non-static) method without declaring it static (non-static) when re-implementing is illegal.
- An *Invocation* of a non-static method from a static context is illegal.
- References like *super.field* or *super.method()* and *this.field* or *this.method()* is illegal from a static context.

5.1.2 The `final` Modifier

The `final` modifier can be used on classes, methods and fields. A final class cannot be sub-classed. A final method cannot be re-implemented, and a final field cannot be re-assigned. Final fields can only be assigned once, by initializing it at declaration time, which is required. Fields in interfaces should be made final automatically, and thus also require an initializer.

Things to Check

- Classes declared `final` cannot be extended in a *ClassDecl*.
- Since `final` fields cannot be re-assigned they must be initialized at declaration time (*FieldDecl*).
- In an *Assignment*, a final field cannot be re-assigned. The same holds for *UnaryPostExpr* and *UnaryPreExpr* like ++ and - -.
- A *MethodDecl* representing a re-implementation of a method is only legal if the method is not already declared `final` in the super class.

5.1.3 The *abstract* Modifier

You should only implement checks associated with the `abstract` modifier if you are implementing **Espresso+**. The `abstract` modifier can be applied to methods and classes. If a method does not have a body, it must be declared `abstract`, and if a class contains one or more `abstract` methods it must be declared `abstract` itself. A class with no `abstract` methods can be declared `abstract`, but keep in mind that abstract classes cannot in instantiated. Interfaces are a special instance of an abstract class where **all** methods are abstract. Interfaces cannot be instantiated as they are technically abstract. An abstract method can of course not be declared final (if it could it would exist without a body and without a possibility of ever giving it one!) or static.

Things to Check

- If a method (*MethodDecl*) does not have a body it should be declared `abstract`.
- If a method (*MethodDecl*) is `abstract` then the class is appears in must be `abstract` as well.
- An `abstract` method cannot be `final`. Nor can a method in an interface.
- An `abstract` class cannot be instantiated (*New*).
- An `abstract` method cannot be `static`.

5.2 *private* and *public*

The modifiers `private` and `public` are opposites. Everything can be declared `public` or `private`, except classes, they can only be `public`. When something is declared public, it can be accessed from anywhere; where as when something is declared private the rules are a little more stringent. Private methods can only be accessed from within the same class as they are defined. Private constructors may only be called by methods or other constructors within the class, that is, they cannot be invoked through a call to *new*. Private fields can only be accessed from within the class as well. The `private` and the `public` modifiers cannot be combined, and if neither is specified it is assumed that the class, method, constructor, or field is `public`.

Things to Check

- An explicit constructor invocation (*CInvocation*) through the *super* keyword cannot access a private constructor in the super-class.
- When accessing fields though a *fieldRef*, private fields can only be accessed if the target of the field reference is of the same type (class) as that of the context. That is, for *object.f* the type (class of *object*) must be the same as the class of the current context.
- An invocation (*Invocation*) of a private method is only legal if the current context is the same type (class) containing the method to be invoked.
- When instantiating a class (though the use of the keyword `new`), the constructor called cannot be `private`.

5.3 Work List

You must implement a new visitor which you can find in the file `ModifierChecker.java`. Each section above outlines the things you must check.

- Determine which *visitXXX()* methods change the *currentConext* and the *leftHandSide* variables and implement them.
- Determine which *visitXXX()* methods require any modifier checking and implement them.
- Check that your compiler works correctly, that is, that all the good **Espresso/Espresso+** files are accepted and that all the bad ones are rejected with good error messages.

5.4 Espresso+

If you are implementing **Espresso+** then you must implement all the checks that have to do with the *abstract* modifier. Recall the `abstract` modifier can only be applied to classes and methods. The things to check for for **Espresso*** are minimal. For example, since the length of an array to the parser looks like a reference to a field in a class or object, we need to make sure we cannot assign to $< array - ref >$ *.length*. Nor can you increment or decrement it using `++` or `--`.

5.5 Test Files

You will find a directory of test files in `/home/CSC460/Espresso/Tests/Phase5/`. These test programs are the ones we will be using throughout the project. You are encouraged to make your own test programs in the same way, and place them in `/home/CSC460/Espresso/StudentTests/`.

In `/home/CSC460/Espresso/Tests/Phase5/BadTests/` we have provided you with one test file for each possible error message you can get by running the reference implementation.

Use this as a reference point for the things you need to check in your traversal trough the tree, and to get an idea of what you must do if you encounter the cases.

To determine whether your compiler is performing correctly, check it against the output from my compiler which you can execute using the `espressocr` script.

5.6 Obtaining the Handout Code

As usual, use the `espressou install 5` command to install phase 5.

5.7 Running the Reference Implementation

In order to check your compiler against my version of the **Espresso** compiler, you can use the `espressocr` script. That will submit your file to my reference implementation and return the output.

5.8 Pre-supplied Code and Testing

The pre-supplied code for this phase is my solution to phase 4 plus the `ModifierChecker.java` skeleton file. The file that you will be doing all your work in is `ModifierChecker.java`.

When you hand in your project we will use the `build.xml` file and the `ant` command to make your project, and the `espressoc` script to run your compiler. If we cannot do that we will not mark your project, i.e., you get 0 on this phase. We will do the following to test your compiler:

```
ant clean
ant
espressoc <testfile.java>
```

so please make sure this process works.

5.9 Due Date

The fourth phase of the project is due ______________________________ before midnight.

Chapter 6

Phase 6—Code Generation

6.1 Introduction

In this final phase of the **Espresso** project you will be generating *Jasmin* assembler code. This assembler code can be translated into class files by executing the *Jasmin* assembler, and then run like any other class file using the Java Virtual Machine. In this phase you will implement more visitors in your compiler.

Each **Espresso** class will in time produce a `.j` file named the same as the class. During this phase, the code is generated in memory (one *Instruction* object per Java byte-code instruction) and held in an array in a *ClassFile* object; each *ClassDecl* is associated with one *ClassFile* object which holds the instructions. Such a *ClassFile* object is then written to the `.j` file through a call to the *writeFile* method in the `WriteFiles.java` file.

The `CodeGenerator` directory contains seven different files:

- `AllocateAddresses.java` — Visitor which allocates addresses for locals and parameters (You will have to do some work in this file.)
- `CodeGenerator.java` — A small class that in turn calls the code generator for each class.
- `GenerateCode.java` — The actual code generator; you will have to do most of your work in this file.
- `GenerateFieldInits.java` — This visitor traverses the tree to generate code for field initializers. Remember field initializers appear either at the beginning of a constructor or at the beginning of the static initializer (depending on whether it is static or not). I have already written the code for this visitor, so you do not need to do any work in this file, but take a look at it anyways!
- `Generator.java` — This file contains a lot of important helper functions that you can use throughout the code generation phases. You should not make any changes to this file, but you should study it carefully as it will help you greatly if you know what the methods in this file do.
- `Java.java` — A special visitor called from `Phase6.java`; this visitor is only used to do some name rewriting for *Object*s, *Thread*s and *Runnable*s.
- `WriteFiles.java` — Code for writing the Jasmin (`.j`) files (the *writeFile* is only called from the *generate* method in the `CodeGenerator.java` file.

You will implement 2 new visitors: *AllocateAddresses* and *GenerateCode*. You should familiarize yourself with the rest of the files though.

The Java Virtual Machine is stack based, that is, it does arithmetic on a stack, so in order to add two numbers you must put them on the stack and then call the appropriate add method, which means that expressions should have code generated for them in postfix notation. The 2 operands are removed, and the result is placed on the stack; this result can then be stored in a local variable or into a field. Remember expressions always leave their results on the stack.

6.2 Instructions

Let us start by looking at the `Instruction` directory; this directory contains 18 different files. Each Jasmin (and thus Java Bytecode) instruction is covered by one of these files (each file contains one class). For example, all the Jasmin instructions without any arguments like for example `iadd` (for adding two integer values on the stack) is represented by an object of *Instruction* class. We can create such an instruction like this:

new *Instruction*(*RuntimeConstants.opc_iadd*)

The file *RuntimeConstants.java* contains all the operand codes of all the instruction. Every time you create an instruction, not matter which one, you have to pass in the instruction's op code. Take a look in this file to see what op codes there are. The class *SimpleInstruction* are instructions that have a single integer parameter like for example `iload 7` or `bipush 34`. Each file contains a list of the instructions represented by that class.

6.3 The Jasmin Directory

The `Jasmin` directory contains the `ClassFile.java` file, which you should take a look at. The only method you really need is the *addInstruction* method. It is used to add instrutions to a classfile (or rather, it is used to add instructions to the current method). A new method is created when visiting a constructor or a method or a static initializer (in the code generator). The other notable file in this directory is the `RuntimeConstants.java`, which contains the constants used to create instruction objects.

6.4 Code Generation Pass

Once we have allocated addresses to all the locals and the parameters (more about that later), we can start the actual code generation phase.

For each *ClassDecl* a new *Generator* is created (this happens in `CodeGenerator.java`; this generator will create a new *ClassFile* object which is made available in the *GenerateCode* class's *classFile* variable. So to add an instruction to a the current method (or constructor/static initializer), simply do the following:

classFile.addInstruction(....);

If you look in *visitMethodDecl* in the *GenerateCode* visitor you can see a call to the *startMethod()*, which starts a new method in the class file.

In the following subsections we look at some of the techniques needed for generating correct code.

6.4.1 Expressions

Some expression can be statements. The complete list is *Assignment* , *Invocation, New, UnaryPreExpression* and *UnaryPostExpression.* This means that the return value of these expressions is not used, and thus left on the stack. This can cause a stack overflow, so you must add an extra `pop` instruction if one of these expressions were used as a statement. Examples:

- `a = 7`
  Returns the integer value 7.

- `new Foo()`
  Returns a reference to a *Foo* object.

- `f()`
  Returns a value of what ever type *f* returns (could be void, then no value is returned).

- `i++`
  Returns the value of `i` before it is incremented.

All such expression statements (represented by an *ExprStat*) must make sure to remove this extra value on the stack by issuing a `pop` instruction. The only exception is if we are dealing with an invocation of a method that returns void. Remember that an assignment returns the value of the assigned value, thus allowing constructions like `a = b = c = 7`. I have completed this code as it is a little complicated and technically is handled in two different places, so do not worry about this code (apart from looking at it and understanding it)

6.4.2 Name Expressions

Earlier a *NameExpr* could represent 3 different things: The use of a local or parameter, or a class name, or a field reference to a field without a target. As mentioned before, I have implemented the *Rewrite* visitor (which you can find in the Utilities directory). This visitor uses the *myDecl* variable in the *NameExpr* to determine if the name expression really is a reference to a field without a target. This will be the case if the *myDecl* points to *FieldDecl*. If this is the case, the *NameExpr* is replaced by a *FieldRef* where the target is either *this* (if the field is non-static) or the name of the class (if the field is static). This rewrite of course means that when traversing the tree, you might not have the exact same tree as the one that was printed in phase 2. You might see a number of *This* and *NameExpr* nodes visited that were not in the original parse tree.

6.5 Assembly Instructions

As stated, we will be generating *Jasmin* assembler code—this code is very close to the mnemonic byte code specified in the JVM specification, so you should look at that. In the following I will point out the differences. Take a look at the JVM specification[1], especially chapter 6. Also consult the user manual and reference material on the *Jasmin* homepage[2].

Most of the *Jasmin* instructions are very similar to the regular JVM byte codes, so you can get all the info you need from chapter 6 in the JVM online reference.

However, you won't be doing the actual printing directly to the Jasmin file; as explained you will generate an array of objects of type *Instruction* (or its subclasses), which then in time will get printed automatically. Therefore you do not need to worry about the correct spacing in the output file etc., you just need to generate the correct instrutions.

Since we do not have the notion of a constant pool in the code that we generate, a few of the instructions in our assembly language differ ever so slightly from their native counterparts. The changed instructions are all the ones that refer to an entry in the constant pool. We simply specify the values of these entries directly in the instruction. Here are examples and explanations to these instructions:

Take a look at the following list of instructions just to get an idea of what they look like; also, note which classes are associated with each of the instructions, if any.

[1]`http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html`
[2]`http://jasmin.sourceforge.net/`

- `.class <modifiers> <name>`
  Defines a class:
  - `.class` — reserved word for defining a class.
  - `<modifiers>` — which ever modifiers the class might have, i.e., `public`, `private`, `final`, `abstract` etc.
  - `<name>` — the name of the class.
- `.interface <modifiers> <name>`
  Defines an interface, the syntax is like that of the `.class`
- `.super <name>`
  Specifies the name of the super class, if the class does not have a super class we use `java/lang/Object`.
- `.implements <name>`
  Specifies that the class implements an interface named `<name>`— one of these lines is needed per interface implemented.
- `.method <modifiers> <name> <signature>`
  Defines a method:
  - `.method` — reserved word.
  - `<modifiers>` — list of the modifiers of the method (i.e., `public`, `private` etc.)
  - `<name>` — the name of the method.
  - `<signature>` — the complete signature of the method, i.e.,
    `(<param signature>)<return type signature>`

  An abstract method in *Jasmin* assembler does not have any body.
- `.end method`
  Specifies the end of a method.
- `.field <modifiers> <name> <signature> = <value>`
  Defines a field:
  - `.field` — reserved word for defining a field.
  - `<modifiers>` — the fields modifiers.
  - `<name>` — the name of the field.
  - `<signature>` — the signature of the type of the field.
  - `= <value>` — optional initial value of the field, can be an integer, a quoted string or a decimal—only used for `final` fields.
- `.limit stack <integer>`
  Specifies the maximum size of the stack at run-time for this method. we just use 50.
- `.limit locals <integer>`
  Specifies the number of local variables (including the parameters) needed for executing this method. This you will have to count.
- `<label name>:`
  Defines a label.
- `getfield <class>/<field name> <signature>`
  `getstatic <class>/<field name> <signature>`
  Gets a field from either an object or a class. To use `getfield` a reference to an object of type `<class>` must be on the stack. The corresponding instruction is a *FieldRefInstruction*.

- `putfield <class>/<field name> <signature>`
  `putstatic <class>/<field name> <signature>`
  Exactly the same as `getfield/getstatic`, just puts a value from the stack into a field. Both object reference and value must be on the stack. The corresponding instruction is a *FieldRefInstruction*.

- `invokenonvirtual <class>/<method>/<signature>`
  Invokes a method in an object. This way of invoking methods is only used for constructors. A reference to the object the method is to be executed in must be on the stack. The corresponding instruction is a *MethodInvocationInstruction*.

- `invokevirtual <class>/<method>/<signature>`
  Like `invokenonvirtual`. This way of invoking methods is used for all methods except constructors and interface methods. The corresponding instruction is a *MethodInvocationInstruction*.

- `invokeinterface <interface>/<method>/<signature>` n
  Invokes a method defined in an interface. Remember even though interfaces cannot be instantiated you can still invoke methods in them, the run-time system will make sure to call the correct method in the object passed as the interface type. `n` holds the number of actual parameters pass to the method, that is, the number of actually parameters on the stack. The corresponding instruction is the *InterfaceInvocationInstruction*.

- `new <class>`
  Creates a new object and places a reference to it on the stack. The corresponding instruction is the *ClassReferenceInstruction*.

- `lookupswitch`
  ```
  <key 1> :  <label 1>
  <key 2> :  <label 2>
  ...
  <key n> :  <label n>
  default :  <label default>
  ```

  Matches the item on the top of the stack (integer values only) to the keys and jumps to the appropriate label. The default label must be last, and make sure to generate one that jumps all the way to the end if no default label was given in the program. The corresponding instruction is the *LookupSwitchInstruction*.

6.6 Signatures

The different types we use in **Espresso** are summarized in the following table and their internal representations are shown:

Type	Signature
byte	`B`
char	`C`
String	`Ljava/lang/String;`
void	`V`
short	`S`
int	`I`
long	`J`
float	`F`
double	`D`
classes/interfaces	`L<class name>;`
boolean	`Z`
array	`[<base type>`

If a class is named A then its corresponding type is `LA;`. Methods also have signatures. The signature of a method looks like $(P_1P_2P_3...P_n)R$ where P_i and `R` are types. An example could be:

```
void foo(A a, boolean b, int c, D d) {
  ...
}
```

which has signature `(LA;ZILD;)V`. `Type.java`, `PrimitiveType.java`, `MethodDecl.java` and `ConstructorDecl.java` all have methods to deal with signatures.

6.7 The Generator Class

Check the `Generator.java` file—it has some useful functions that you will want to make use of:

- Methods for getting and setting break and continue labels.
- Methods for setting, getting and incrementing addresses.
- *getClassFile()*—gets the classfile associated with this generator.
- *endMethod()*—adds the appropriate load and return instructions to assure that we do not fall off the end of the code.
- *public void dataConvert(Type from, Type to)*—generates code to automatically cast between numeric types when needed.
- *public void loadInt(String s)*—write the appropriate code for loading an integer constant, depending on its value.
- *public void loadLong(String s)*—write the appropriate code for loading a long value.
- *public void loadFloat(String s)*—write the appropriate code for loading a float value.
- *public void loadDouble(String s)*—write the appropriate code for loading a double value.
- *public void loadString(String s)*—write the appropriate code for loading a string value.
- *public String getLabel()*—returns a new unused label.
- *getStoreInstruction()*—returns the appropriate store instruction based on type and address and whether the store is into an array.

- *getLoadInstruction()*—returns the appropriate load instruction based on type and address and whether the load is into an array.
- *getArrayStoreInstruction()*—returns the appropriate instruction to store a value into an array.
- *getArrayLoadInstruction()*—returns the appropriate array load instruction.
- *getBinaryAssignmentOpInstruction()*—returns the appropriate binary operator instruction based on operator and type, should only be used for `<op>=` operators.
- *dup()*—return one of argument 2 or 3 based on the type of argument 1. Typically this method is called with arguments 2 and 3 being `dup` and `dup2` or `dup_x1` and `dup2_x1` or `dup_x2` and `dup2_x2`. Argument 3 is returned if argument one represents a double or a long type.
- *loadConstant1()*—loads the value 1 based on its type.
- *addorSub()*—generates an add or sub instruction according to the type.
- *getOpcodeFromString()*—takes in a string representing an instruction and returns the corresponding op-code as found in *RuntimeConstants.java*.

6.8 Assembling the *Jasmin* (`.j`) File

The `.j` file you generate must be translated into a `.class` file so you can run it using SUN's JVM (java). This is done by calling the assembler:

```
./jasmin test.j
```

which will generate `test.class`. The *Jasmin* assembler is located in a jar file in the `src/Utilities` folder and can be invoked by using the jasmin script in your project root directory.

6.9 Running Your Code

Java requires that the method its JVM starts when loaded is a public static method called *main* that returns *void* and takes in an array of strings. Even if you have not implemented arrays, your main should always look like this: **public static void** *main*(**String**[] *args*).

Remember, Java imports and Espresso imports are completely different. Java's import mechanism is a lot fancier than that of Espresso. Java uses packages, whereas Espresso is 'flat' at the top level. That is, all classes whether imported or not, live in the same class table. However, you can do some importing in Espresso. If you want to import a class, this class must be located in the include directory that you specify in the call to the compiler (by default this directory is called `Include`. I have placed a few predefined files in this directory, most notably the *Io.java* file, which, if you look inside it, is mostly empty! Also, the file *System* has a static field of type *Io*, which means that in Espresso programs we can actually write things like *System.out.println(...')*. When implementing classes in the include directory, their compiled (with the Java compiler) must be placed in the `Lib` directory; that is where the run-time (by default) will look for them. You can see the actual implementation of the it Io.java class in the file *Lib\Io.java*. Note, this file could not have been compiled with the Espresso compiler, as it contains syntax that our compiler does not accept.

I suggest you use the `espresso` shell script to run your code; it adds the correct class-paths for things to work correctly. If you do not want to use the `Lib` directory for your import implementations, then you need to specify the new directory on the class path to Java.

6.10 Coding Hints

When generating code you want to make sure you take it in smaller steps. Start out with a class like

```
class Small {
}
```

and make sure you generate code that looks like:

```
.class public Small
.super java/lang/Object

.method public <init>()V
    .limit stack 50
    .limit locals 1
    aload_0
    invokenonvirtual java/lang/Object/<init>()V
    return
.end method
```

That is, that you get the correct class and super headings, that you get a standard constructor etc. Then start adding fields and methods and so on.

6.10.1 Boolean Expressions

Consider the following class:

```
class Small {
  Small() {
    if (2 == 3)
      return;
    else
      return;
  }
}
```

and the code returned when compiled:

```
.class public Small
.super java/lang/Object
.method public <init>()V
        .limit stack 50
        .limit locals 1
        aload_0
        invokenonvirtual java/lang/Object/<init>()V
        iconst_2
        iconst_3
        if_icmpeq L3
        iconst_0
        goto L4
L3:
        iconst_1
L4:
        ifeq L1
        return
```

```
                goto L2
        L1:
                return
        L2:
                return
        .end method
```

As you can see, it is definitely not optimal. It seems that the check has been implemented a strange way, i.e., 0 (false) is put on the stack if the boolean expression is false and 1 (true) if it is true. You could easily optimize this to just jump to either the then part or the else part. The problem here is that the boolean expression (a *BinaryExpr*) was generated by a call to *visitBinaryExpr()*, which did not know that it was part of an if statement. Since a *BinaryExpr* is an expression, and must thus leave its value on the top of the stack, the code starting with `iconst_2` and ending with `L4:` was generated in *visitBinaryExpr()* and not in *visitIf()* . This is the reason that the *if, while, do, switch* and *for* statements might look a little weird, but unless you want to repeat a lot of code this is the easiest way to do it.

6.10.2 Return Values

All methods must have a return statement at the end of its code. This is needed to assure that we do not execute code past the end of the method. We did not check that all possible executions of a method would lead to a return, so as a little hack we always put a return at the end of a method. For a *void* method just use return, for a method returning int you push 0 on the stack and return using `ireturn`. The same happens for booleans. For methods returning objects you push *null* on the stack and use `areturn`. If your method has an explicit return at the end you will notice code that looks like

```
            ...
            ireturn
            iconst_0
            ireturn
        .end method
```

This is OK, even though the last two instructions are never executed. An optimizer could easily be written to take care of this, but we will not bother with that for this assignment.

6.10.3 Assigning Addresses

Each local or parameter needs to have an address. An address represents a storage location for a variable in the activation record for a running method. Address 0 always holds a the address of the current object (i.e., *this*), and can be loaded onto the stack using the `aload_0` instruction. The *Generator* class has a variable called *address* that can be used to hold the next available address. This should be reset to 1 every time a new method is compiled, unless it is a static method, then we start at address 0 (except for main, this one still starts at 1). *ParamDecl* and *LocalDecl* each has a field called *address* that will hold the address assigned to that variable.

Write a new visitor called *AllocateAddresses* that traverses the parse tree and assigns addresses to all locals and parameters. Note that a field does not have an address as it is not stored in the activation record, but rather in the object itself. Fields can be accessed using the `putfield` and `getfield` assembly instructions, where as local variables and parameters are accessed using `Xload` and `Xstore` (where `X` is `i` for integers and booleans and `a` for objects etc.). Also keep count of the number of locals and parameters you have assigned addresses to for each method as this number is needed when generating the method definition.

Remember, at address 0, an object reference to 'this' is stored. This means that the first available address will be 1. However, for static methods and static initializers, there exists no 'this'. In this situation we start at address 0.

6.10.4 Generating Code for Field Declarations

If a field declaration has an initializer, then you need to generate code for it. Unfortunately you cannot generate the initializer code when you visit the *FieldDecl* on your way through the tree. The reason is that this visit to the *FieldDecl* should only generate code that defines the field, an example could be:

```
public class Foo {
  public int bar = 100;
  public final double baz = 98.0;
  Foo() {
  }
}
```

The code generated for field declaration when the 2 *FieldDecls* are visited should be:

```
.field public bar I
.field public final baz D = 98.0
```

The parts are:

- `.field`: tells the assembler that the class has a field.
- `public` and `final` are modifiers for the fields
- `bar` and `baz`: the name of the fields are *bar* and *baz*.
- `I` and `D`: the types are integers and doubles (fields can never be of void type).

We only generate the initializer code for a field now if the field is `final` and the initializer is a simple value (i.e., `final int a = 3;`. If we generated code for other initializers at this time the code would not be inside any method, and thus never be executed (and the assembler would most likely complain as well). The trick is to execute all initializers for fields in the constructor, so when generating code for a constructor make sure you generate the code for the field initializers at this time.

This is most easily done by implementing a new visitor called *GenerateFieldInits()*, re-implementing *visitFieldDecl()*, and call this visitor when you are generating the constructor (either in *visitConstructorDecl()* if there is a constructor or in *visitClassDecl()* if no constructor is specified). Note that any class must have a constructor, so if none has been specified you must create a default constructor. The reason is that a call to `java.lang.Object.<init>` must be invoked as the very first thing when an object is created.

Initialization of static fields cannot take place in the constructor, if it did, the static fields would have their values overwritten every time we created an object from that class. Instead they are placed in the static initializer (also known as the static block—which can explicitly be specified as `static {...}`). If no static initializer block was specified one must be generated if there are static fields that need to be initialized. This method is called `<clinit>` and has signature `()V`. This method cannot be explicitly called, but is automatically called by the JVM when the class is loaded.

6.10.5 Break and Continue

One check that we have not done yet is assuring that `break`s and `continue`s always appear within loops. This is not hard but should be done. Remember to generate labels for where a break and where a continue should jump to when you start generating code for loops. Hint: Use another visitor and a 2 flags that tell you if you are inside or outside a construct that you can break/continue in.

6.10.6 Input/Output

As explained before, we can easily do input and output if we import the *System* class. Remember, in Java, classes are imported on demand if fully qualified, but Espresso does not do that, so the import is needed if you want to use the class. **import** *System* will import the *System* class, which in turn imports the *Io* class so you can do output like you do in Java. Input is equally easy, simply call the appropriate *System.in.readInt()* method to read an integer.

6.10.7 Assignments and Unary Post/Pre Expressions

Generating code for assignments and unary post/pre expressions is not as straight forward as you might think (especially not if you want to generate good tight code). One of the things that makes this especially challenging is the `<op>=` assignment operators (e.g., `+=` along with array references. Consider a simple assignment like this:

$a+=4;$

If a is a local/parameter or a field, we could rewrite the assignment to be $a = a + 4$; and then just recall the code generator. You might think that we can do the same with an assignment like this one:

$aa[f(x)]+=4;$

That is, we can construct this assignment instead: $aa[f(x)] = aa[f(x)] + 4$ and then just call the code generator again. That is not the case; what if the call $f(x)$ had a side effect, e.g., adding 1 to a field. In the first case 1 would be added to the field, but 2 would be added in the second version. Even worse, the second call to $f(x)$ in the second version might not even return the same value as the first call. So, if we want to generate code that is impervious to all these side effects we have to be a little more careful, and that involves duplicating various values and references on the stack.

Assignments

Let us start with the assignment and then return to the unary expressions later on. The code generation can be broken down into 6 smaller steps, which should make it fairly simple to implement. (Let us refer to the Assignment node as *as* in the following.

1. If the left hand side of the assignment is a FieldRef or an ArrayAcccessExpr we need to generate a reference to it. For a FieldRef we simply visit the target and for an ArrayAccessExpr we first visit the target and then the index. This will leave either an object reference or an array reference and an index on the stack.

2. This step is only done if the assignment operator is of the form `<op>=`, i.e., not for simple assignment (`=`). We need to generate code to load the value of the left hand side and then we need to do some duplicating. Let us consider the three different possibilities:

   - $as.left()$ is a *FieldRef*: If the field is non-static we need to duplicate (`dup`) the object reference on top of the stack before we issue a `getfield` instruction. If the field is static, there is no reference to duplicate, and we simply issue the `getstatic` instruction.
   - $as.left()$ is a *ArrayAccessExpr*: Since we need to store a value back into the array, we need to duplicate both the reference to the array as well as the index, that is, a reference and an integer value. This can be done in a very clever way: we can use the `dup2` instruction to duplicate **one** two-word value or 2 one-word values (in this case a one-word reference and a one-word integer value). This is followed by the appropriate array load instruction (`Xaload`).

- *as.left*() is a *NameExpr*: This case is simple; just load the value of the local or parameter on to the stack with one of the `XloadY / Xload_Y` instructions.

3. Now we simply visit the right hand side (*as.right*()) of the assignment, and remember to update `RHSofAssignment`. After the visit, we should call *dataConvert* to covert the type of the value on top of the stack to the type of the left hand side.

4. This step is like step 2 only for assignments with an `<op>=` operator. Now we need to perform the operation associated with `<op>`: examples include `isub`, `dadd`, etc.

5. If this assignment with which we are working itself is a right hand side of another assignment we need to do some clever duplicating of values and possibly moving them around on the stack. Again, we do different things depending on the type of the left hand side:
   - *as.left*() is a **FieldRef**: If the field is static simply duplicate the value on top of the stack using either a `dup` or a `dup2`. If the field is non-static we use `dup_x1` or `dup2_x1`.
   - *as.left*() is a **ArrayAccessExpr**: Use a `dup_x2` or `dup2_x2`.
   - *as.left*() is a **NameExpr**: Use a `dup` or `dup2`.

6. The final step is to generate the instruction to store the value on top of the stack back into the left hand side. Remember, step 1 generated the reference already.
   - *as.left*() is a **FieldRef**: Issue either a `putfield` or a `putstatic`.
   - *as.left*() is a **ArrayAccessExpr**: Issue the appropriate `Xastore`.
   - *as.left*() is a **NameExpr**: Issue the appropriate `XstoreY` or `Xstore_Y`.

Unary Post Expressions

To generate code for a *UnaryPostExpression* we can follow the following steps; there are three different scenarios: local/parameters, field references, and array references.

- If the expression in a *NameExpr* do the following
  - Visit it.
  - If the expression is of integer type you can use the `iinc` instruction in increment the local or the parameter.
  - Otherwise (if the type is not integer) do the following four steps:
    - Duplicate the value on the stack.
    - Load the appropriate constant 1.
    - Generate the appropriate add or sub instruction.
    - Store back into local or parameter using `Xstore Y` or `XstoreY`.
- If the expression in a *FieldRef* do the following:
  - If it is a static field, do the following:
    - Visit the target. (this might seem like a strange thing to do as the field reference is to a static field, but the target expression could have side-effects, so it must have code generated for it).
    - We might now need to issue a *pop* instruction because the execution of the target might have left a useless reference value on top of the stack. This should happen unless the target is the name of a class and the field is static.
    - Use `getstatic` to get the value from the appropriate class on to the stack.

    * Generate code to duplicate the value on the stack; we need to do that because the value that should be on the stack after we are done with the post increment or decrement operation should be the value of the field **before** the operation.
  - If it is a non-static field, do the following:
    * Visit the target to get a reference to the object in which the field resides.
    * Issue a *dup* instruction.
    * Generate a `getfield` instruction to load the value of the field on to the stack.
    * Generate code to duplicate the value on the stack using either `dup_x1` or `dup2_x1`.
  - Generate code that loads the appropriately typed constant 1 on to the stack.
  - Generate the appropriate addition or subtraction instruction.
  - Generate a *pufield* instruction to store the value back into the field.

- If the expression in an array reference, do the following:
  - Visit the target.
  - Visit the index expression.
  - Issue a `dup2` instruction.
  - Generate the appropriate array load instruction (you might want to use the *getArrayLoadInstruction()* method in the *Generator* class.
  - Issue a `dup_x2` or a `dup2_x2` instruction.
  - Generate code that loads the appropriately typed constant 1 on to the stack.
  - Generate the appropriate addition or subtraction instruction.
  - Generate the appropriate array store instruction (you might want to use the *generateArrayStoreInstruction()* method in the *Generator* class.

Unary Pre Expressions

A unary pre expression is rather close to the unary post expression when it comes to implementation. The major difference is in the placement of the duplicate instruction that duplicates the value that the expression evaluates to. In the post expression, the value used is the value before the incrementation/decrementation, whereas in the pre expression, the value used is that of the expressions after the incrementation/decrementation.

Another slight difference is the fact that the pre expression has a couple of extra operators like `-`, `+`, `!`, and `~`. These extra operators are easy to deal with, so let us do that first.

- If the operator is `-`, then visit the the expression and issue the appropriate negation operator. Here is an example of how I did it:
  String *instString* = *up.expr().type.getTypePrefix()* + "neg";
  classFile.addInstruction(**new** *Instruction*(*Generator.getOpCodeFromString*(*instString*)));
  Note, sometimes when dealing with the instructions that take no parameters but are type specific, it is easier to generate the instruction in a string and then use the *getOpCodeFromString* method.

- If the operator is `+`, then just visit the expression.

- If the operator is `~`, then visit the expression, and if the type is integer, issue a `iconst_m1` instruction followed by a `ixor` instruction, else if the type is not integer, use `ldc2_w` to load the constant -1 and issue an `lxor` instruction.

- If the operator is `!`, then visit the expression, issue a `iconst_1` instruction and an `ixor` instruction. Remember, a bit-wise negation can be done using the constant 1 and an exclusive or operator.

Those were the operators that did not require any storing of the result value, but we still have to deal with the `++` and the `--` operator, so here we go (again we have to distinguish between the three different types: local/parameter, field reference, or array reference.

- If the expression is a *NameExpr*, do the following:
  - If the type of the expression in integer, use the `iinc` instruction to increment or decrement the local or the parameter, then visit the expression.
  - If the type of the expression is not integer, do the following:
    - Visit the expression.
    - Load the appropriately typed constant 1.
    - Generate the appropriately types addition or subtraction instruction.
    - Duplicate the value using either `dup` or `dup2`.
    - Store the value back using either `XstoreY` or `Xstore_Y`.
- If the expression is a *FieldRef*, do the following:
  - If the field is not static, visit the target and generate a `getfield` instruction.
  - If the field is static, visit the target and generate a `pop` instruction if the target is not a class name - you can use this line if you like: !(*fr.target*() **instanceof** *NameExpr* && (((*NameExpr*)*fr.target*()).*myDecl* **instanceof** *ClassDecl*)) to determine if a `pop` should be issued. Finally, generate a `getstatic` instruction.
  - Generate the code to generate the appropriately typed constant 1 on the stack.
  - Generate the appropriately typed addition or subtraction.
  - Generate a dup. If the field is non-static use either `dup_x1` or `dup2_x1`, and if the field is static use `dup` or `dup2`.
  - Finally generate a `pufield` or `putstatic` instruction.
- If the expression is a *ArrayAccessExpr*, do the following:
  - Visit the target and the index expression.
  - Generate a `dup2` instruction. This duplicates the array reference as well as the index (both computed in the previous step).
  - Generate the appropriate array load instruction.
  - Generate code to load the appropriately typed value 1 onto the stack.
  - Generate the appropriately typed addition or subtraction.
  - Duplicate the stack using either `dup_x2` or `dup2_x2`. This places a copy of the newly computed value below the array reference and index already on the stack.
  - Generate the appropriate array store instruction. This will consume the top three values on the stack and expose the value that was placed here in the step above.

Here is an example of how the stack should behave for the following expression: `--`*a[5]*; if the array *a* has base type integer and is located at address 3:

Instruction	Stack
`aload_3`	... ref
`iconst 5`	... ref index(5)
`dup2`	... ref index(5) ref index(5)
`iaload`	... ref index(5) value
`iconst_1`	... ref index(5) value 1
`isub`	... ref index(5) value-1
`dup_x2`	... value-1 ref index value-1
`iastore`	... value-1

You should try making this kind of chart for all the techniques described above to get a better understanding of why your code should look like this!

I suggest you read the section on implementing arrays in Espresso (the next chapter) to better understand the array instructions.

6.11 Obtaining the Handout Code

Use the `espressou install 6` to install the handout code for phase 6.

6.12 Running the Reference Implementation

In order to check your compiler against my version of the **Espresso** compiler, you can use the `espressocr` script. That will submit your file to my reference implementation and return the output. It will generate a number of `.rj` files just like your compiler generates `.j` files. If you got it right they should match.

6.13 Pre-supplied Code and Testing

The handout code includes my solution to phase 5 in addition to the new files in the `CodeGenerator` directory.

When you hand in your project we will use the `build.xml` file to build your project, and the `espressoc` script to run your compiler. If we cannot do that we will not mark your project, i.e., you get 0 on this phase. We will do the following to test your compiler:

```
ant clean
ant
espressoc <testfile.java>
```

so please make sure this process works. For this phase it is not enough that your compiler works, i.e., that it generates some sort of code. The code generated must be good too. This means that it should be valid assembler code that can be run through *Jasmin*, and produce a working `.class` file that can be run using the regular Java Virtual Machine.

6.14 Due Date

The final phase of the project is due ____________________________ before midnight.

Chapter 7

Adding Arrays to Espresso

In this final section, we turn to arrays; a real programming language has arrays, so let us add them to Espresso. This is actually a fairly straight forward task with changes in only a few places in the code. Examples of files that need changing are:

- Adding rules for the terminals `'['` and `']'` to the flex file (`espresso.flex`).
- Adding grammar rules and action to create parse tree nodes to the cup files (`espresso.cup`).
- Adding visitor methods to `Visitor.java`.
- Adding visitor methods to `TypeChecker.java`.
- Adding visitor methods to `CodeGenerator.java`.
- A few changes in `Type.java`.

The new parse tree nodes required as well as entries in the *Visitor* pattern are already present after phase 2, and if you decide to implement arrays, you will of course have to make changes to the flex and the cup file as well as what might be requires in the type checker and the code generator. The changes to the grammar (and thus to the cup file) are already outlined in the grammar specification in chapter 1.
In Espresso, like in Java, we do not have static arrays, so all arrays must be allocated with the **new** keyword. Therefore, an array variable is *always* declared without dimensions. Examples include:

int[ ] *a*;
MyClass[ ][ ] *myAs;*

An array is allocated using the **new** keyword. An allocated can be with an initializer, in which case there are no dimensions specified:

int[ ][ ] *a* = **new int**[ ][ ] {{1, 2}, {3, 4}};

Arrays can also be allocated without an initializer, in which case at least one dimension must be given. When one ore more dimensions are given, they must appear before empty dimensions, and once a dimensions is empty, no more non-empty dimensions may be given.

int[ ][ ] *a* =**new int**[10][10]
int[ ][ ] *a* =**new int**[10][]

are both legal array allocations, whereas

int[ ][ ][ ] *a* =**new int**[10][ ][10]

is not legal.
Java allows arrays to be declared in two different ways: the [ ] can be placed on the type or on the variable name. All three of the following declarations declare a 2 dimensional array of integers:

int[ ][ ] *ints*;
int[ ] *ints*[ ];
int *ints*[ ][ ];

To handle the type where the array brackets appear on the variable name we need to be a little tricky; what we will do is add a field to the *Name* parse tree node called *arrayDepth*, which counts the number of brackets appearing on the variable name, and then when creating the parse tree node for either a *LocalDecl*, *ParamDecl* or a *FieldDecl*, we check if the name of the local/parameter/field has any array brackets associated with it, and if it has, we either replace the associated type with the appropriate array type, or, of the type is already an array type, we create a new array type with the appropriate dimensions (the dimensions of the type plus the number of brackets on the name).

This way, we always construct declarations with array types of the correct dimension, and no array information (brackets) are associated with the name. Note, we do this 3 different places in the CUP file.

7.1 Changes to the Grammar

There are three major changes to the grammar: The declaration, the initalization (array literals) and the array access. As mentioned before, these changes are already incorporated in the grammar in chapter 1, but let us list them separately here.
First we need to add the new terminals for array indexing. The only two new terminals are `[` and `]`:

```
   terminal Token LBRACK, RBRACK; // [ ]
```

7.1.1 Types, Values and Variables

We need to add the array type to the right hand side of the reference type, and add a new rule for the array type.

```
// ----------------------------------
// 19.4) Types, Values, and Variables
// ----------------------------------

// Type:  Type :> AST
 reference_type  :  class_or_interface_type
                 |  array_type

// Type:  ArrayType :> Type :> AST
 array_type  :  primitive_type dims
             |  name dims
```

7.1.2 Field Declarations

In this section we have to add the array initializer to the variable initializer.

```
// -----------------------------------
// 19.8.2) Field Declarations
// -----------------------------------

// Type:  Expression :> AST
 variable_initializer  :  expression
                       |  array_initializer
// Type:  Name :> AST
 variable_declarator_id  :  name
                         |  variable_declarator_id '[' ']'
// -----------------------------------
// 19.10) Arrays
// -----------------------------------

// Type:  ArrayLiteral :> Expression :> AST
 array_initializer  :  '{' variable_initializers_opt '}'

// Type:  Sequence(Expression :> AST) :> AST
 variable_initializers_opt  :  variable_initializers
                            |  ε
// Type:  Sequence(Expression :> AST) :> AST
 variable_initializers  :  variable_initializer
                        |  variable_initializers ',' variable_initializer
// -----------------------------------
// 19.12) Expressions
// -----------------------------------

// Type:  Expression :> AST
 primary  :  primary_no_new_array
          |  array_creation_expression
// Type:  Expression :> AST
 primary_no_new_array  :  literal
                       |  'this'
                       |  '(' expression ')'
                       |  class_instance_creation_expression
                       |  field_access
                       |  method_invocation
                       |  array_access
// Type:  NewArray :> Expression :> AST
 array_creation_expression  :  'new' primitive_type dim_exprs dims_opt
                            |  'new' class_or_interface_type dim_exprs dims_opt
                            |  'new' primitive_type dims array_initializer
                            |  'new' class_or_interface_type dims array_initializer


// Type:  Sequence(Expression :> AST) :> AST
 dim_exprs  :  dim_expr
            |  dim_exprs dim_expr
// Type:  Expression :> AST
 dim_expr  :  '[' expression ']'

// Type:  Sequence(Expression:> AST) :> AST
```

```
 dims_opt  :  dims
           |  ε

// Type:  Sequence(Expression :> AST) :> AST
 dims  :  '[' ']'
       |  dims '[' ']'

// Type:  ArrayAccessExpr :> Expression :> AST
 array_access  :  name '[' expression ']'
               |  primary_no_new_array '[' expression ']'

// Type:  Expression :> AST
 left_hand_side  :  name
                 |  field_access
                 |  array_access
```

7.2 New Parse Tree Nodes

There are four new parse tree nodes that deal with arrays:

- `ArrayType.java` — Representing an array type. For example, **int**[ ][ ][ ] is an *ArrayType* with base type **int** and dimension 3.

- `ArrayLiteral.java` — Representing an array literal, like {{1, 3} ,{4, 5, 6}}.

- `ArrayAccessExpr.java` — Representing an array access like $e_1[e_2]$. where e_1 is an expression of array type and e_2 is an integral expression.

- `NewArray.java` — Representing a new array creation. An example is **new int**[10] or **new double**[ ][ ]{{1.0, 2.0}, {3.0, 4.0}}.

7.3 Type Checking Arrays

Let us briefly consider what type checking is needed for the four new parse tree nodes.

7.3.1 ArrayLiteral

Type checking array literals is not quite as easy as other literal values. We know that for example 3.4 is of double type, but what type is { }? It is an array literal that could be assigned to the type **int**[ ]. It could also be assigned to the type **int**[ ][ ], etc. Therefore, we cannot determine the definite type of an array literal, but we can determine if an array literal can be assigned to an array type by writing a fairly straightforward recursive procedure *arrayAssignmentCompatible*, which takes in an array type and an array literal. The only things we need to remember is that any array type can be assigned the value { }, and when we get down to a non-array item the value must be assignment compatible with the base type of the array type. Table 7.1 illustrates this check.

For this reason, there is no need to even visit an *ArrayLiteral* in the type checker; instead, what we have to do is perform the checking as described above in the *NewArray* (besides, this is the only place where an *ArrayLiteral* can appear.

7.3.2 ArrayType

Since an *ArrayType* is already a type, all we have to do is return it.

Level	Literal	Type	
	$\{\underline{\{\{30, 40\}, \{\}\}}, \underline{\{\{\}, \{50, 60\}\}}, \underline{\{\}}\}$	**double**[ ][ ][ ]	
[0]	$\{\underline{\{30, 40\}}, \underline{\{\}}\}$	**double**[ ][ ]	
[1]	$\{\underline{\{\}}, \underline{\{50, 60\}}\}$	**double**[ ][ ]	
[2]	{}	**double**[ ][ ]	√
[0][0]	$\{\underline{30}, \underline{40}\}$	**double**[ ]	
[0][1]	{}	**double**[ ]	√
[0][1][0]	30	**double**	√
[0][1][1]	40	**double**	√
[1][0]	{}	**double**[ ]	√
[1][1]	$\{\underline{50}, \underline{60}\}$	**double**[ ]	
[1][1][0]	50	**double**	√
[1][1][1]	60	**double**	√

Table 7.1: Example of *arrayAssignmentCompatible*.

7.3.3 NewArray

The first thing we check for a *NewArray* expression[1] is that all the non-empty dimensions are of integral type, that is, you cannot have something like this: **new int**[12.5][][]. If there is an initializer (in which case all the dimensions will be empty), then we simply call the *arrayAssignmentCompatible* method we described above.

7.3.4 ArrayAccessExpr

Recall, an *ArrayAccessExpr* consists of an expression and **one** index in [ ] (There will be one parse tree node for each set of [ ]), so all we have to check here is that the expression is of array type, and naturally we have to set the type of the entire parse tree node, which is the base type of the array type if the type is only one dimension deep, otherwise its a new array type with the same base type as the expression with one dimension less.

7.4 Code Generation for Arrays

For how to deal with assignments and unary expressions involving arrays you should consult the chapter on code generation; I have added this information there.

7.4.1 Jasmin Array Related Instructions

Creating a New Array

There are three different instructions for creating a new array.

[1]Note, though a *NewArray* parse tree node represents an expression, a new array creation cannot appear as a statement expression.

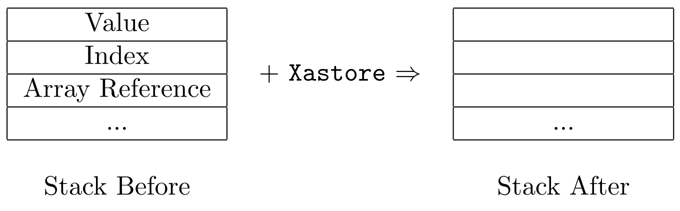
`newarray <type>`	where `<type>` is a primitive type like `int`. E.g.: `newarray int` (for **new int**[10]).
`anewarray <reference type>`	This instruction has two different applications: • Creating a single dimensional array of references type. E.g.: `anewarray A` (from **new A**[10]). • Creating a multidimensional array where only one dimensional expression is given (for the first dimension). E.g.: `anewarray [[I` (from **new int**[10][ ][ ]).
`multianewarray <type> <n>`	Creating a multidimensional array of `<type>` with `<n>` non-empty dimensions. E.g.: `multianewarray [[I 2` (from **new int**[3][4]) and `multianewarray [[[I 2` (from **new int**[3][4][ ]).

Loading and Storing Values in Arrays

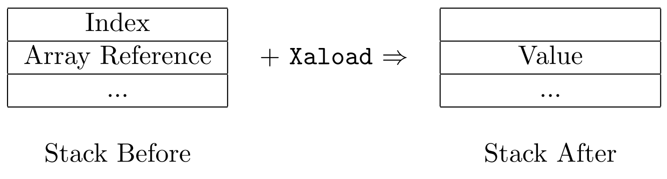
The following table summarizes the loading and storing instructions for arrays:

`aaload`	Loads a value of reference type from an array onto the operand stack.
`iaload`	Loads a value of integer type from an array onto the operand stack.
`daload`	Loads a value of double type from an array onto the operand stack.
`faload`	Loads a value of float type from an array onto the operand stack.
`laload`	Loads a value of long type from an array onto the operand stack.
`aastore`	Stores a value of reference type from the operand stack into an array.
`iastore`	Stores a value of integer type from the operand stack into an array.
`dastore`	Stores a value of double type from the operand stack into an array.
`fastore`	Stores a value of float type from the operand stack into an array.
`lastore`	Stores a value of long type from the operand stack into an array.

For each of the `load` instructions, the operand stack must hold a reference value to an array object as well as a valid index.



For each `store` instruction, the operand stack must hold a reference value to an array, a valid index into this array as well as the value to store in the array. If the index is invalid, and *ArrayIndexOutOfBound* exception will be thrown by the virtual machine.



The code generation of assignments, unary pre and post expressions can be a little tricky. The description of how to proceed with these can be found in the general code generation chapter.

7.5 Array Specific Parse Tree Nodes

Let us briefly look at what is needed to generate code for the three array specific parse tree nodes. The *ArrayAccess* is very easy; since all the complicated stuff related to assigning to arrays and post/pre incrementing them is dealt with (and described) elsewhere, an *ArrayAccess* is as easy as just visiting the target and the index.

An *ArrayLiteral* is a little more involved. We have to issue either a `newarray` or `anewarray` (the former if the array literal is one-dimensional, the latter if it is multi-dimensional), so the first thing to do is to load an integer value signifying the length of the array. Now issue either a `newarray` or `anewarray` as described above. Each element in the array (whether primitive or array) needs to be stored into the appropriate place with an `Xastore` instruction. We can loop though the children of the *ArrayLiteral* and do the following:

- Duplicate the array reference on top of the stack.
- Load the integer array index.
- Visit the expression (the child).
- Issue a *Xastore* instruction.

Now let us turn to the *NewArray* parse tree node. It can represent a number of different array creations (with or without initializers, one, two or multi-dimensional etc). The code generation can progress as follows:

- If we have a simple one-dimensional array creation like **new int**[10] then first visit the dimensional expression and issue a `newarray` if the base type is a primitive, or a `anewarray` otherwise.
- If we have a multi-dimensional array creation with m ($m \geq 0$) empty dimensions like **new** *A*[10][20][30] or **new int**[10][20][][] (see the next line too!) in order, visit the index expressions, and then issue a `multinewarray`.
- If we have a multi-dimensional array with only **one** of the dimensions having an expression like **new int**[10][] or **new** *B*[100][][], we visit the dimension expression and then issue a `anewarray` (`anewarray [I` for the first and `anewarray [[[B` for the second.
- Finally if there is an initializer, just visit it. It will generate the correct new arrays.

Those were the instructions for how to generate code for the three array specific parse tree nodes. Again, remember, in the code generation chapter, information about how to generate code for assignments to arrays as well as unary post and pre expressions.

7.6 Changes to `Type.java`

The first change we need to make is to add an *isArrayType()* method:

```
public boolean isArrayType() {
    return (this instanceof ArrayType);
}
```

The *getTypePrefix()* method in `Type.java` must have a case added for arrays:

```
public String getTypePrefix() {
    if (this.isClassType() || this.isNullType() || this.isStringType() || this.isArrayType())
        return "a";
```

```
⋮
```

And finally we need to add a line to the *parseSignature()* method:

```
    case '[': s = s + " ["; break;
```

In Java, the type prefix for arrays is an `a` like it is for other reference types. The signature of an array is a `[` followed by its base type.

7.7 Other Changes

We naturally must also add visitor methods for all four new parse tree nodes in `Visitor.java`:

```
public Object visitArrayAccessExpr(ArrayAccessExpr ae) {
        return ae.visitChildren(this);
}

public Object visitArrayLiteral(ArrayLiteral al) {
        return al.visitChildren(this);
}

public Object visitArrayType(ArrayType at) {
        return at.visitChildren(this);
}

public Object visitNewArray(NewArray ne) {
        return ne.visitChildren(this);
}
```

In addition, if we want nice printing of the parse tree we need to add these four additional methods to the file `PrintVisitor.java`:

```
/** ARRAY ACCESS EXPRESSION */
public Object visitArrayAccessExpr(ArrayAccessExpr ae) {
        System.out.println(indent(ae.line) + "Array Access Expression:");
        indent += 2;
        super.visitArrayAccessExpr(ae);
        indent -=2;
        return null;
}

//** ARRAY LITERAL */
public Object visitArrayLiteral(ArrayLiteral al) {
        System.out.println(indent(al.line) + "Array Literal:");
        indent += 2;
        super.visitArrayLiteral(al);
        indent -=2;
        return null;
}

//** ARRAY TYPE */
public Object visitArrayType(ArrayType at) {
        System.out.println(indent(at.line) + "Array Type:  ");
        for (int i=0;i<at.getDepth();i++)
                System.out.print("[]");
        indent += 2;
        super.visitArrayType(at);
        indent -=2;
        return null;
}

public Object visitNewArray(NewArray ne) {
        System.out.println(indent(ne.line) + "New Array:");
        indent += 2;
        super.visitNewArray(ne);
        indent -=2;
        return null;
}
```

7.8 Obtaining the Handout Code

Everything you need to finish the array part of the compiler for each phase is already present, so you need no special code. If you do not implement arrays, you can simply ignore these extra classes, files, and methods.

7.9 Running the Reference Implementation

In order to check your compiler against my version of the Espresso compiler, you can use the `espressocr` script. That will submit your file to my reference implementation and return the output.

Appendix A

Do You Really Know Java?

Take a look at the program below and try to see if you can predict the output **without** actually running it.

```
class MyClass {
    public int value;
    public MyClass(int v) {
        value = v;
    }

    public int getValue() {
        return value;
    }
}

interface A {
    public int field = 10;
    public MyClass mc = new MyClass(10);
    int f();
}

class B implements A {
    public int f() {
        return field;
    }
}

class C extends B {
    public int field = 20;
    public MyClass mc = new MyClass(20);
    public int f() {
        return field;
    }
}

class Main {
    public static void main(String args[ ,]) {

        A a1 = new B();
        A a2 = new C();
```

```
            B b1 = new B();
            B b2 = new C();
            C c = new C();

            System.out.println(a1.f() +    + a1.field +    + a1.mc.getValue());
            System.out.println(a2.f() +    + a2.field +    + a2.mc.getValue());
            System.out.println(b1.f() +    + b1.field +    + b1.mc.getValue());
            System.out.println(b2.f() +    + b2.field +    + b2.mc.getValue());
            System.out.println(c.f() +    + c.field +    + c.mc.getValue());

        }
    }
```

Try filling in the table before you run the program.

Object	$f()$	$field$	$mc.getValue()$
$a1$			
$a2$			
$b1$			
$b2$			
c			

Appendix B

Writing Libraries for Espresso

It is fairly easy to write your own libraries for Espresso. I have already done so with the System library for performing input and output. Let us take a look at how I managed to incorporate input and output into the Espresso system. Doing I/O requires the runtime to interact with the operating system, and we do not have those possibilities in Espresso, but we do in Java, and since we will be executing the Espresso class files in the JVM we can make use of any Java library. The easiest way to do that is to write a new Espresso file with the correct method headers but empty implementations, and then write a similar Java file with the same headers, but with a correct Java implementation. The Espresso 'header' file should be placed in the `Include` directory, and the compiled Java file (the class file) should be placed in the `Lib` directory. The `espresso` shell script then automatically sets the Java class path to point to the `Lib` directory. Let us see how to achieve this with the *System* library. In order to make input and output look like real Java (i.e., *System.out.println*), we need a `System.java` file in the *Include* directory that looks like this:

```
import Io;

public class System {
    static Io out;
}
```

We see that the *System* class contains only one static file, namely *out*, which is of type *IO*. The *IO* class looks like this:

```
class Io {
    static public void print(byte b) { }
    static public void print(short s) { }
    static public void print(char c) { }
    static public void print(int i) { }
    static public void print(long l) { }
    static public void print(float f) { }
    static public void print(double d){ }
    static public void print(String s){ }
    static public void print(boolean b){ }
    static public void println(byte b) { }
    static public void println(short s){ }
    static public void println(char c) { }
    static public void println(int i) { }
    static public void println(long l) { }
    static public void println(float f){ }
    static public void println(double d){ }
    static public void println(String s){ }
    static public void println(boolean b){ }

    static public int readInt() { }
    static public long readLong() { }
    static public float readFloat() { }
    static public double readDouble() { }
    static public String readString() { }
}
```

We can see that this file would not compile with Java (missing returns in the read* methods), but Espresso is ok with it.

I have implemented the *import* statement to automatically search the `Include` directory for a file of the name being imported (it is worth noting that the Espresso import does not work like the Java import) and then create a new parser for it, read it, and build the appropriate parse trees. There will be no checking to do because all the files have empty methods. Now to get things working we must provide real Java files in the `Lib` directories for both files:

```
public class System {
    static Io out;
}

import java.io.*;
import java.lang.Integer;
import java.lang.Double;

class Io {
    static BufferedReader in = new BufferedReader(new InputStreamReader(java.lang.System.in));

    static public void print(byte b)  { java.lang.System.out.print(b); }
    static public void print(short s) { java.lang.System.out.print(s); }

    ...

    static public double readDouble() throws IOException  {
        return Double.parseDouble(in.readLine());
    }
    static public String readString() throws IOException  {
        return in.readLine();
    }
}
```

B.1 Adding a Math Library

Let us design a very simple Math library that you can extend yourself if you like. First we design an Espresso class with the appropriate method headers (Let us just add the *abs* method):

```
public class Math {
     public static double abs(double x) {

     }
}
```

and and place the file `Math.java` in the `Include` directory. The implementation file (also called `Math.java`) should be place in the `Lib` directory and compiled **with the java compiler**:

```
public class Math {
        public static double abs(double x) {
                return (x ¡ 0) ? -1*x : x;
        }
}
```

and voila, we have a very simple math library which we can use by placing an **import** *Math*; statement at the top of the Espresso file.

Appendix C

Espresso Fundamentals

Type	Size	Minimum Value	Maximum Value
boolean		`true` or `false`	
byte	8 bits	-128	127
short	16 bits	-32,768	32,767
char	16 bit	/u0000	/uFFFF
int	32 bits	-2,147,483,648	2,147,483,647
long	64 bits	-9,233,372,036,854,775,808	9,233,372,036,854,775,807
float	32 bits	-3.4E+38 w/7 significant digits.	3.4E+38 w/7 significant digits
double	64 bits	1.7E+308 w/15 significant digits	1.7E+308 w/15 significant digits

Primitive types in Espresso

Operator	Description	Example	Result
!	logical NOT	!*a*	`true` if *a* is `false`, `false` if *a* is `true`.
&&	logical AND	*a* && *b*	`true` if *a* and *b* are both `true`, `false` otherwise.
||	logical OR	*a* || *b*	`true` if *a* or *b* or both are `true`, `false` otherwise.

Espresso Logical Operators

Operator	Meaning
==	equal to
! =	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equalto

Espresso Equality and Reltaional Operators

Operator	**Description**
∼	bitwise NOT
&	bitwise AND
|	bitwise OR
ˆ	bitwise XOR
<<	left shift
>>	right shift
>>>	right shift with zero fill

Espresso Bitwise Operators

From	**To**
byte	short, int, long, float, or double
short	int, long, float, or double
char	int, long, float, or double
int	long, float, or double
long	float or double
float	double

Espresso Widening Conversions

From	**To**
byte	char
short	byte or char
char	byte or short
int	byte, short, or char
long	byte, short, char, or int
float	byte, short, char, int, or long
double	bytem short, char, int, long, or float

Espresso Narrowing Conversions

Modifier	**Classes and Interfaces**	**Methods and Variables**
default (no modifier)	Same as `public`	Same as `public`
`public`	Visible anywhere	Visible anywhere
`private`	Not used in Espresso	Not visible by any other class.

Espresso Visibility Modifiers

In an espression, operators at a lower precedence level are evaluated before those at a higher level. Operators at the same level are evaluated according to the specified association.

Precedence level	**Operator**	**Operation**	**Associates**
1	.	Object member reference.	L to R
	(*parameters*)	Parameter evaluation and method invocation.	
	++	Postfix increment.	
	−	Postfix decrement.	
2	++	Prefix increment.	R to L
	−	Prefix decrement	
	+	Unary plus.	
	-	Unary minus.	
	∼	Bitwise NOT.	
	!	Logical NOT.	
3	**new**	Object instantiation.	L to R
	(*type*)	Cast.	
4	*	Multiplication.	L to R
	/	Division.	
	%	Remainder.	
5	+	Plus	L to R
	-	* Subtraction.	
6	<<	Left shift.	L to R
	>>	Right shift.	
	>>>	Right shift with zero fill.	
7	<	Less than.	L to R
	<=	Less than or equal to.	
	>	Greater than.	
	>=	Greater than or equal to.	
	instanceof	Type comparison.	
8	==	Equal.	L to R
	! =	Not equal.	
9	&	Bitwise AND.	L to R
10	ˆ	Bitwise XOR.	L to R
11	|	Bitwise OR.	L to R
12	&&	Logical AND.	L to R
13	||	Logical OR.	L to R
14	? :	Ternary conditional operator.	R to L
15	=	Assignment.	R to L
	+ =	Addition, then assignment.	
	− =	Subtraction, then assignment.	
	∗ =	Multiplication, then assginment.	
	/ =	Division, then assignment.	
	% =	Remainder, then assignment.	
	<<=	Left shift, then assignment.	
	>>=	Right shift (sign), then assignment.	
	>>>=	Right shift (zero), then assignment.	
	& =	Bitwise AND, then assignment.	
	ˆ=	Bitwise XOR, then assignment.	
	| =	Bitwise OR, then assignment.	

Appendix D

Espresso Reserved Words

abstract (modifier)

Syntax

[*modifier*] [**abstract**] class *class-name*
    [extends *class-name*]
    [implements *class-name* [, *class-name* ...]] { ... }

or

[*modifier*] [**abstract**] *data-type method-name* ([*parameter-declarations*]) ;

Description

The **abstract** modifier can be applied to both classes and methods. In the case of classes, applying the **abstract** modifier prevents the class in question from being instantiated. **Abstract** methods must be implemented in a subclass. In both cases, the idea is that a subclass will provide the completed definition of the class or **abstract** methods.

Example

```
public abstract class ClassBodyDecl extends AST {
  public ClassBodyDecl(AST a) {
    super(a);
  }

  // isStatic() and getname() must be implemented by
  // appropriate subclasses
  public abstract boolean isStatic() ;

  public abstract String getname() ;
}
```

abstract (continued) (modifier)

Tips

- If one or more **abstract** methods are present in a class, then the class must be **abstract**.
- A constructor cannot be modified with the **abstract** reserved word.
- **Abstract** Methods cannot also be private methods.
- Interface classes are, by default, **abstract** in nature.

boolean (data type)

Syntax

[*modifier*] **boolean** *variable-name* [= *initial-value-expression*] ;

Description

The reserved word **boolean** is used to declare one or more variables of the **boolean** data type (containing a “true” or “false” value).

Examples

```
// Declares a boolean named done and sets it initial value to false
boolean done = false;
```

Tips

- When declaring a **boolean** value, you can optionally set the initial value of the variable by following the variable name by a = and then the initial value expression.
- Multiple **boolean** variables can be declared in the same statement by following each variable name with a comma.
- Booleans cannot be cast to any other type, nor can any other type be cast to boolean.

break (control)

Syntax

break;

Description

The **break** statement stops execution of the enclosing switch, while, do, or for loop and continues execution following the enclosing statement.

Example

```
// The break statement prevents the execution from executing the
// code associated with the following labels.
switch (gpa) {
case 1:
   Io.println("Poor");
   break;
case 4:
   Io.println("Excellent");
   break;
}
```

```
found = false;
for (int i=0; i<max; i++) {
   if (bag[i] == target) {
      found = true;
      break;
   }
}
```

Tips

- If a **break** statement is not enclosed by a switch, while, do, or for loop, a compilation error will occur.
- If the **break** statement is absent from the end of a case clause, execution will continue through the next **case** label and its statements.

byte (data type)

Syntax

[*modifiers*] **byte** *variable-name* [= *initial-value-expression*];

Description

The reserved word **byte** us used to declare one or more variables of the **byte** data type. In Espresso, bytes are 8-bit signed values. The minimum value of one byte is -128; the maximum is 127.

Example

```
// Declares a byte variable named alpha and sets its initial value to 19.
byte alpha = 19;

// Declares two byte variables, beta and gamma, and sets gamma's initial
// value to -5.
byte beta, gamma = -5;
```

Tips

- When declaring a **byte** variable, you can optionally set the initial value of the variable by following the variable name with an equal sign and the initial value expression (as shown in brackets in the syntax statement).
- Multiple **byte** variables can be declared in the same statement by following each variable name with a comma (see the second part of the example).

case (control)

Syntax

```
switch ( expression ) {
        // One or more case clauses of the form
        [ case constant-expression:
                statement(s);
                [ break; ]
        ]

        [ default:
                statement(s);
                [ break; ]
        ]
}
```

Description

The **case** label is used within switch statements to define executable blocks of code to be executed when the switch's expression evaluates to the **case** label's constant expression.

Example

```
char letter = 'i';
String characterType = "unknown";

// Switch on the "letter" variable.
switch (letter) {
  // if "letter" is 'a', 'e', 'i', 'o', or 'u', set the type to "vowel"
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
    characterType = "vowel";
    break;

  // Otherwise set the type to "consonant"
  default:
    characterType = "consonant";
    break;
}
```

case (continued) (control)

```
int value = 3;

// Switch on the value variable
switch(value) {

  // If value is 1, increment by one.
  case 1:
    value++;
    break;

  // If value is 3, increment by two
  case 3:
    value += 2;
    break;
}
```

Tips

- The constant expression must be a char, byte, short, or an integer literal.
- Constant expressions within the same switch statement must be unique. One or more statements, terminated with a break statement, generally follow **case** labels. If the break statement is absent, execution will continue through the next **case** label and its statements (if present); **case** labels can also cascade through this mechanism, as shown in the first example.
- The default label can be used to define a block of statements (like a **case** label) that are executed if no **case** label matches the constant expression.

char (data type)

Syntax

[*modifiers*] **char** *variable-name* [= *initial-value-expression*];

Description

The reserved word **char** is used to declare one or more variables of the **char** data type (which contains character data values). In Espresso, chars are 16-bit unsigned values representing characters from the Unicode character set. Only 128 of the more than 65,000 character symbols that Unicode supports are traditionally used for programming and printing information in the English language.

Example

```
// Declares a char variable named firstLetter and sets its initial value to 'p'.
char firstLetter = 'p';

// Declares two char variables, alpha and beta, and sets beta's initial value to '*';
char alpha, beta ='*';
```

Tips

- When declaring a **char** variable, you can optimally set the initial value of the variable by following the variable name with an equals sign and the initial value expression (as shown in the brackets in the syntax statement).
- Multiple **char** variables can be declared in the same statement by following each variable named with a comma (see the second part of the example).

class (class-related)

Syntax

```
[ modifiers ] class class-name
            [ extends class-name ]
            [ implements class-name [ , class-name ... ] ] {
      // Fields, constructors, static initializers and methods are defined here.
}
```

Description

The **class** reserved word is used to define the implementation of a class in the Espresso language.

Example

```
public class Student {
  // Stores the name and address of the student as Strings
  private String name;
  private String address;

  // Returns the name of this student
  public String getName() {
    return name;
  }
}
```

Tips

- A **class** can inherit from another **class**—this is known as inheritance. Inheritance occurs by using the extends reserved word. A **class** can inherit only from one parent **class**.
- An object can be created from the definition of a **class** by using the new reserved word.
- The definition of a **class** can be modified with one or more of the following: public, abstract, final.

continue (control)

Syntax

continue;

Description

The **continue** statement terminates processing of the current iteration of a loop at the point of the statement. This behaviour is similar to that of a break statement, but rather than terminating the loop completely, the **continue** statement evaluates the loop condition and iterates again through the loop if the condition yields a `true` value.

Example

```
while (alpha < 5) {
  // Increment alpha and terminate this iteration of the loop.
  alpha++;
  if (alpha > 0)
    continue;

  // The increment of beta is never executed.
  beta++;
}
```

Tips

- The **continue** statement can also be used in do and for loops.
- A compile-time error will occur if a continue statement is not enclosed by a do, for or while loops.
- Care should be exercised when using the **continue** statement, as it may make source code more difficult to read.

default (control)

Syntax

default:

Description

The **default** label is used within switch statements to define executable blocks of code to be executed in the event that the switch's expression fails to evaluate to any existing case label's constant expression.

Example

```
int value = 3;

// Perform a switch on the value variable, defined earlier.
switch (value) {
   // If value is 3, increment by two.
   case 3:
      value += 2;
      break;

   // The default case; increment value by one.
   default:
      value++;
      break;
}
```

Tips

- A **default** label is not followed by a constant expression, unlike the case labels also present in switch statements.
- The **default** label is generally listed last in the switch block. This helps distinguish it from the case labels. However, if the default label appears earlier in the list of labels, it does not impede the ability to correctly match case labels found later in the switch statement block.

do (control)

Syntax

do
statement;
while (*boolean-expression*);

Description

The **do** reserved word is used to construct a loop that executes the specified statement one of more times until the expression following the while reserved word evaluates to a `false` value.

Example

```
// Continually loop and increase the value of alpha until alpha's value exceeds 5.
do
   alpha += beta;
while (alpha < 5);
```

Tips

- The statement portion of the **do** loop always executes at least once.
- If the boolean expression does not resolve to a boolean result (`true` or `false`) a compile-time error is generated.
- To execute multiple statements as the body of the **do** loop, enclose the statements to execute in braces, creating a statement block.
- Be sure that the boolean expression changes as part of the body of the loop. If the value of the expression does not change, the loop will be performed endlessly. This situation is known as an endless loop.
- Keep in mind that the boolean expression can comprise multiple boolean expressions, each joined by logical operators.
- A **do** loop may contain other loops (**do**, for, while), thereby creating nested loops.
- Break and continue statements can be used in **do** loops.
- A common mistake is to forget to insert a semicolon (;) following the boolean expression.

double (data type)

Syntax

[*modifiers*] **double** *variable-name* [= *initial-value-expressions*];

Description

The reserved word **double** is used to declare one or more variables of the **double** data type. In Espresso, doubles are 64-bit signed values. The approximate minimum value of a **double** is -1.7E+308 and the approximate maximum value is 1.7E+308. A value of the **double** data type can contain up to fifteen significant digits.

Example

```
// Declares a double variable named alpha and sets it initial value to 45.6964
double alpha = 45.6964
```

```
// Declares two double variables, beta and gamma, and sets gamma's initial
// value to -55.112D.
double beta, gamma= -55.112D;
```

Tips

- When declaring a **double** variable, you can optionally set the initial value of the variable by following the variable name with an equals sign and the initial value expression (as shown in brackets in the syntax statement).
- Multiple **double** variables can be declared in the same statement by following each variable name with a comma (see the second example).
- To reduce confusion between floating point literal values, **double** literals can be appended with either the 'd' or the 'D' character (see the second example).

else (control)

Syntax

```
if ( boolean-expression )
        statement;
[ else
        statement;
]
```

Description

The **else** reserved word is used to provide an alternative statement to execute in the event that an if statement's boolean expression resolves to `false`. When used in conjuncture with the if statement, a programmer can create an execution path for a `true` and a `false` result in an if statement.

Example

```
// Check to see whether the number of students in the course is equal to 25. if so,
// close this section. Otherwise, ensure that the section remains open
if (numberOfStudents == 25)
   classFull = true;
else
   classFull = false;
```

Tips

- The **else** reserved word (and its corresponding statement) is an optional part of the if conditional statement.
- To execute multiple statements as part of the **else** clause of an if statement, enclose the statements to execute with braces, creating a statement block.

extends (class-related)

Syntax

[*modifier*] class *class-name*
[**extends** *class-name*]
[implements *class-name* [, *class-name* ...]] { ... }

or

[*modifier*] interface *interface-name* **extends** *interface-name* { ... }

Description

The **extends** reserved word is used to modify a class's definition indicating that the child class is derived from the named parent class (class name).

Example

```
public class Rectangle extends Shape {
   ...
}
```

Tips

- Inheritance is the concept that one class is derived from another (existing) class.
- The existing class (Shape in the example) is known as the parent class, or superclass. The new class (also known as the derived class; Rectangle in the example) is known as the child class or subclass.
- In the Espresso programming language, a class can inherit from only one parent class. Thus, when using the **extends** keyword, you never name more than one class.
- Both abstract and interface classes can be extended.

final (modifier)

Syntax

[*modifiers*] [**final**] class *class-name*
[extends *class-name*]
[implements *class-name* [, *class-name* ...]] { ... }

or

[*modifiers*] [**final**] *data-type method-name* ([*parameter-declarations*]) { ... }

or

[*modifiers*] [**final**] *data-type variable-name* [= *initial-value-expression*];

Description

The **final** reserved word modifier is used to change the characteristics of classes, methods, and fields. Generally, the use of the **final** reserved word implies that the item is modifies cannot be changed.

Example

```
public final class PI {
  // Sets the value of PI we use in this class;
  final double VALUE = 3.1415;

  // Returns the area of the circle with radius rValue
  public final double getRSquared(double rValue) {
    return VALUE * rValue * rValue;
  }
}
```

Tips

- A class that is declared **final** cannot be subclassed or declared to be abstract.
- A method that is declared **final** cannot be overridden or declared to be abstract.
- A field (class or instance) that is declared **final** cannot be modified and must be initialized at the point of declaration.
- A constructor cannot be modified with the **final** reserved word.

float (data-type)

Syntax

[*modifier*] **float** *variable-name* [= *initial-value-expression*];

Description

The reserved word **float** is used to declare one or more variables of the **float** data type. In Espresso, floats are 32-bit signed values. The approximate minimum value of a **float** is -3.4E+8 and the approximate maximum value is 3.4E+8. A value of the **float** data type can contain up to seven significant digits.

Example

```
// Declares a float variable named alpha and
// sets its initial value to 6.965.
float alpha = 6.965f;

// Declares two float variables, beta and gamma, and
// sets gamma's initial value to -12.345F.
float beta, gamma = -12.345F;
```

Tips

- When declaring a **float** variable, you can optionally set the initial value of the variable by following the variable name with an equals sign and the initial value expression (as shown in brackets in the syntax statement).
- Multiple **float** variables can be declared in the same statement by following each variable name with a comma (see the second example).
- To reduce confusion between floating point literal values, **float** literal values must be appended with either the 'f' or the 'F' character (see the example).

for (control)

Syntax

for (*initializer-expression* ; *test-expression* ; *update-expression*)
    statement;

Description

The **for** reserved word is used to construct a loop that executes the specified statement until the test expression evaluates to a `false` value. Before the first attempted execution of the loop, the initializer is executed (generally used to set loop control variables).

Next, the test expression is evaluated. If the test expression evaluates to a `true` result, the statement of the loop body is executed. Following each iteration of the loop, the update expression is performed and then the test expression is reevaluated. If the test expression evaluates to a `false` result, the loop terminates and control passes to the statement following the loop.

The initializer expression may optionally declare a variable and set its initial value (see the example). If declared in the initializer expression, the variable is accessible only in the body of the loop; it is not accessible outside the loop.

The initializer expression and update expressions may contain several expressions separated by commas. The test expression can comprise multiple expressions joined by zero or more logical operators.

Example

```
// This loop is controlled by alpha (decreasing from 5 to 1). Each time through the
// loop, the value of alpha is decremented and printed.
for (int alpha = 5; alpha > 0; alpha--) {
  lo.print("Alpha's value is: ");
  lo.println(alpha);
}
```

Tips

- To execute multiple statements as the body of the **for** loop, enclose the statements to execute in braces, creating a statements block.
- Be sure that the boolean expression eventually changes as part of the body of the loop. If the value of the expression never changes, the loop will be performed endlessly.
- A **for** loop may contain other loops (do, **for**, while), thereby creating *nested loops*.
- Break and continue statements can be used in **for** loops.

if (control)

Syntax

if (*boolean-expression*)
    statement;
[else
    statement;
]

Description

The **if** reserved word is used to build a conditional statement that may be used if the boolean expression is a `true` value. The statement is executed only if the boolean expression resolves to a `true` result. If the boolean expression is `false` and an else clause is present (following the statement), the else statement is executed.

Example

```
// Check to see whether the number of students in the course is equal to 25. if so,
// close this section. Otherwise, ensure that the section remains open
if (numberOfStudents == 25)
  classFull = true;
else
  classFull = false;
```

Tips

- If the boolean expression does not resolve to a boolean result (`true` or `false`), a compile time error is generated.
- To execute multiple statements when the boolean expression is a `true` value, enclose the statements to execute with braces, creating a statement block.
- It is possible to nest **if** statements (the **if** reserved word, the boolean expression, and the statement). That is, an **if** statement can be executed as a result of an other **if** statement.

implements (class-related)

Syntax

[*modifier*] class *class-name*
    [extends *class-name*]
    [**implements** *class-name* [, *class-name ...*]] { ... }

Description

The **implements** reserved word denotes that a given class provides method implementations of the specified interface class.

Example

```
public class Student implements Undergraduate { ... }
```

Tips

- The **implements** reserved word can be used in conjunction with the extends reserved word to modify a class definition.
- A class can implement multiple interfaces by following the **implements** reserved word with a comma-separated list of interface class names.

instanceof (class-related)

Syntax

reference-name **instanceof** *class-name*

Description

The **instanceof** reserved word is an operator used to validate that a given reference to an object (reference name) is an instance of a specified interface or class (class-name). A boolean result (`true` or `false`) is the result of an operation using the **instanceof** operator.

Example

```
// Verify that the myObject is a Student object. If it is, cast it to a Student
// and print the name of the student.
if (myObject instanceof Student)
   Io.println(((Student)myObject).getName());
```

Tips

- The **instanceof** operator is frequently used to ensure that an object is of a particular type before casting it to that type.

int (data type)

Syntax

[*modifier*] **int** *variable-name* [= *initial-value-expression*]

Description

The reserved word **int** is used to declare one or more variables of the integer data type. In Espresso, integers are 32-bit signed values. The minimum value of an integer is -2,147,483,648 and the maximum value is 2,147,483,647.

Example

```
// Declare an integer variable named alpha and set its initial value to 19.
int alpha = 19;

// Declare two integer variables, beta and gamma, and set gamma's
// initial value to -5.
int beta, gamma = -5;
```

Tips

- When declaring an **integer** variable, you can optionally set the initial value of the variable by following the variable name with an equals sign and the initial value expression (as shown in the brackets in the syntax statement).
- Multiple **integer** variables can be declared in the same statement by following each variable name with a comma (see the second part of the example).

interface (class-related)

Syntax

[*modifier*] **interface** *interface-name* extends *interface-name* {
 constant-definitions;
 abstract-methods;
}

Description

An **interface** defines one or more constant identifiers and abstract methods. A separate class implements the interface class and provides the definitions of the abstract methods. Interfaces are used as a design technique to help organize properties (identifiers) and behaviours (methods) the implementing class may assume.

Example

```
// The ShapeInterface defines two abstract methods for implementation by
// any shape. It also provides a weightMultiplier for use in various
// shape-related calculations.
public interface ShapeInterface {
   public final double weightMultiplier = 69.64;

   public int getArea() ;
   public int getCircumference() ;
}
```

Tips

- A class definition implements an **interface** through the use of the implements reserved word in its definition and through the implementation of the interface's abstract methods.
- An **interface** class cannot be instantiated.
- Methods contained in an **interface** class cannot contain statements. That is, their implementation must be left to the class that implements the **interface**.
- Multiple classes can implement the same **interface**, each providing different definitions of the abstract methods of the **interface**.
- A class can implement more than one **interface** at a time.

long (data type)

Syntax

[*modifier*] **long** *variable-name* [= *initial-value-expression*];

Description

The reserved word **long** is used to declare one or more variables of the **long** data type. In Espresso, longs are 64-bit signed values. The minimum value of a **long** is -9,223,372,036,854,775,808 and the maximum value is 9,223,372,036,854,775,807.

Example

```
// Declares a long variable named alpha and sets its initial value to 19.
long alpha = 19l;

// Declares two long variables, beta and gamma, and sets gamma's initial
// value to -55063L.
long beta, gamma = -55063L;
```

Tips

- When declaring a **long** variable, you can optionally set the initial value of the variable by following the variable name with an equals sign and the initial value expression (as shown in the brackets in the syntax statement).
- Multiple **long** variables can be declared in the same statement by following each variable name with a comma (see the second part of the example).
- To reduce confusion between integer literal values, **long** literal values can be appended with either the 'l' or the 'L' character (see the example).

new (class related)

Syntax

variable-name = **new** ([*parameter-list*]);

Description

The **new** reserved word is used to create a new object (instance) of the specified class name. Generally, the instantiation is performed as the right-hand side of an assignment statement. The instantiation of the class executes the class's constructor and may contain zero of more parameters.

Example

```
// Create a new Student object by calling the default constructor.
Student freshman = null;
freshman = new Student();
```

Tips

- Objects can be instantiated without assigning them to variables. This event most often occurs when creating an object and passing it as a parameter to a method.
- Object instantiation does not necessary need to occur in conjunction with variable declaration. It can be performed later and a reference assigned from the variable to the newly created object (see the example).
- When instantiating an object from a given class (class name), a programmer makes a call to one of the class's constructors (methods with the same name as the class and without a return type). The call to the constructor must match a constructor's parameter list (in order, type, and number of parameters).

private (modifier)

Syntax

[**private**] [*modifier*] *data-type method-name* ([*parameter-declaration*]) { ... }

or

[**private**] [*modifier*] *data-type variable-name* [= *initial-value*] ;

Description

The **private** reserved word is used to modify the visibility of methods, and data fields (Espresso does not support private classes). Methods and data field definitions can be modified with the **private** reserved word thereby preventing them from being accessed by subclasses. Additionally, **private** methods and data fields are inaccessible by any other class.

Example

```
public class Private Example {
  // The privateField data field is not inherited by subclasses and
  // is accessible only by instances of this class.
  private double privateField = 0.0;

  // The incrementField method is not inherited by subclasses and is
  // accessible only by this class.
  private void incrementField() {
    privateField += 3.1415;
  }
}
```

Tips

- The use of **private** visibility helps promote encapsulation of the methods and data fields of a class. The selection of the visibility modifier for a method or a field is an integral decision of the engineering of software.

public (modifier)

Syntax

[**public**] [*modifier*] class *class-name*
    [extends *class-name*]
    [implements *class-name* [, *class-name* ...]] { ... }

or

[**public**] [*modifier*] *data-type method-name* ([*parameter-declaration*]) { ... }

or

[**public**] [*modifier*] *data-type variable-name* [= *initial-value*] ;

Description

The **public** reserved word is used to modify visibility of classes, interfaces, methods, and data fields. When used with the definition of a class, **public** opens accessibility of the class to any other code (though the internal data fields and methods may have other visibility). Classes and interfaces in Espresso can only be declared **public**, if the **public** modifier is not present it when declaring a class or an interface, is automatically assumed. **Public** methods are generally used to provide a service (e.g., accessors and mutators) to other classes. **Public** data fields are generally considered poor programming (unless they are also declared final) because **public** data violates the notion of encapsulation of data with the encompassing class.

Example

```
public class Circle {
  // The value of PI is available to this and other classes but cannot
  // be changed.
  public static final double PI = 3.1415;

  // The static area method returns the area of a circle, given a radius. It
  // uses the defined PI value.
  public static double area(double radius) {
    return PI * radius * radius;
  }
}
```

Tips

- If the constructor for a given class is non-**public**, then only the class itself can make instances of the class.

return (control)

Syntax

return [*expression*];

Description

The **return** reserved word is used alone or with the optional expression to create a statement used to return execution to the calling method. When encountered, execution ceases in the current method and execution is transferred to the statement in the calling method or constructor.

Example

```
// The getPostalCode method returns the postal code value (an integer)
public int getPostalCode() {
   return postalCode;
}
```

```
// The setPostalCode method sets the postal code to the value specified
// and does not return a value.
public void setPostalCode(int value) {
   postalCode = value;
   return;
}
```

Tips

- If an expression is present in a **return** statement, the expression's resulting data type must match the data type specified to be returned according to the method's signature. That is, the expression must match the method's return type.
- Although it is possible to have multiple return statements, it is considered good programming practice to limit methods to containing only one **return** statement.
- If a method does not return any data (the return type is void), a **return** statement need not be present. In the second example, it is present but returns nothing, so its presence does not cause an warnings or errors.
- Constructors can contain a **return** statement; but they must be devoid of the optional expression.

short (data-type)

Syntax

[*modifier*] **short** [= *initial-value-expression*];

Description

The reserved word **short** is used to declare one of more variables of the **short** data type. In Espresso, shorts are 12-bit signed values. The minimum value of a **short** is -32,768 and the maximum value is 32,767.

Example

```
// Declares a short variable named alpha and sets its initial value to 19.
short alpha = 19;

// Declare two short variables, beta and gamma, and sets gamma's initial
// value to -5.
short beta, gamma= -5;
```

Tips

- When declaring a **short** variable, you can optionally set the initial value by following the variable name with an equals sign and the initial value expression (as shown in the syntax statement).
- Multiple **short** variables can be declared in the same statement by following each variable name with a comma (see the second part of the example).

static (modifier)

Syntax

[*modifier*] [**static**] *data-type method name* ([*parameter-declarations*]) { ... }

or

[*modifier*] [**static**] *data-type variable-name* [*initial-value*];

Description

The **static** reserved word is used to modify a method or data field to be a class method (or data field) rather than an instance method (or data field). That is, there will only be one copy of the method or data field for all objects of the class that contains it.

Example

```
public class Course {
  // The static crsPrefix identifier is used in printing the TCNJ prefix.
  private static String crsPrefix = "TCNJ";

  // The getPrefix method returns the static data crsPrefix.
  public static String getPrefix() {
    return crsPrefix;
  }
}
```

Tips

- A **static** method can reference only **static** data and other **static** methods—it had no knowledge of instances of the class that contains it. All instances of a class can access instance data and instance methods as well as the **static** data and **static** methods of the class.
- If a **static** variable's value is modified, the change is seen by all of the instances of a class.
- A constructor cannot be modified with the **static** reserved word.
- Constant data fields (declared with the final reserved word) are often modified with the **static** reserved word to preserve memory. (Multiple copies of a value that cannot be changed are wasteful.)
- **Static** methods are generally used to provide a service to other classes. For example, the Io class, which contains a number of writing and reading methods are considered service methods.

super (class-related)

Syntax

super.*method-name* ([*parameters*]);

or

super.*variable-name*

or

super([*parameters*]);

Description

The **super** reserved word can be used in the non-static methods or constructors of a class to refer to its parents class or constructors in the parent class. It may also be used to refer to data fields in a superclass.

Example

```
public class Beta extends Alpha {
   // The Beta constructor calls a constructor from Alpha that accepts one
   // integer value. The super (without a methods name) method call refers to a
   // constructor in the Alpha class. The count variable contained in the
   // Alpha class is also incremented.
   public Beta(int value) {
      super(value);
      super.count++;
   }

   // This Beta constructor calls the getRandomValue method from the Alpha
   // class. It will also print the value of the count data field from the super class.
   public Beta() {
      Io.println(super.getRandomValue());
   }
}
```

super (class-related)

Tips

- You can use the **super** reserved word to call a specific constructor in the parent class. If present, the call to the superclass's constructor must be the first statement in the subclass's constructor (as shown in the example).
- Methods or data fields existing in the parent class (which have been overridden in a subclass) can be accessed by the subclass using the **super** reserved word. Use of this reserved word helps to specify which class a method class or field reference is referring to.

String (data type)

Syntax

[*modifier*] **String** *variable name* [= *initial-value*];

Description

The **String** data type is considered an atomic, or built-in, type in Espresso (This is not the case in Java though). **String** constants are enclosed in quotation marks (").

Example

```
public String greeting = "Hello World!";
```

Tips

- **String** values can be printed and read using the Io class provided.
- The empty **String** is denoted "";
- Note that the **String** data type is the only data type that is capitalized.

switch (control)

Syntax

```
switch ( expression ) {
        // One or more case clauses of the form
        [ case constant-expression:
                statment(s);;
                [ break; ]
        ]

        [ default:
                statement(s);
                [ break; ]
        ]
}
```

Description

The **switch** statement is used to provide an option for selecting and executing one of a number of paths. The selection is based on the result of the expression and the **switch** statement containing a case label for the path that matches the expression's result.

Example

```
char letter = 'i';
String characterType = "unknown";

// Switch on the "letter" variable.
switch (letter) {
  // if "letter" is 'a', 'e', 'i', 'o', or 'u', set the type to "vowel"
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
    characterType = "vowel";
    break;

  // Otherwise set the type to "consonant"
  default:
    characterType = "consonant";
    break;
}
```

switch (continued) (control)

```
int value = 3;

// Switch on the value variable
switch(value) {

  // If value is 1, increment by one.
  case 1:
    value++;
    break;

  // If value is 3, increment by two
  case 3:
    value += 2;
    break;
}
```

Tips

- The **switch** expression must evaluate to a byte, char, int, or short value.
- Any statement within the **switch** statement block must be labeled by a case or default label.

this (class related)

Syntax

this.*data-field*

or

this.*method-name*([*parameter-list*])

or

this([*parameter-list*])

Description

The **this** reserved word in Espresso is used within an object to refer to itself. It is most often used in constructors and instance methods to refer to instance data fields that might have the same name as the formal parameters of the constructor or the method. The **this** reserved word can therefor be used to clarify which variable (the formal parameter or instance data field) you wish to reference. It can also be used to invoke other constructors within the same class. As with the super reserved word, if a constructor is invoked using the **this** reserved word, this invocation must occur on the first line of the constructor.

Example

```
public class Section {
  // The instance variable sectionNumber and SectioEnrollment keep track of
  // key information about each instance of a course's section.
  private int sectionNumber;
  private int sectionEnrollment;

  // The Section constructor accepts values for the section number and
  // the size of this section.
  Section(int sectionNumber, int sectionEnrollment) {
    this.sectionNumber = sectionNumber;
    this.sectionEnrollment = sectionEnrollment;
  }
}
```

Tips

- The **this** reserved word is technically a reference to the current object. it can also be used as a parameter to a method. Example: enrollment.add(**this**);

void (data type)

Syntax

[*modifier*] **void** *method-name* ([*parameter-list*]) { ... }

Description

The **void** reserved word is used to denote the absence of a data value to be returned at the conclusion of a method's execution.

Example

```
// The setName method sets the character's name to the value of the cName
// parameter, but returns nothing to the calling method.
public void satName(String cName) {
   characterName = cName;
}
```

Tips

- If you use the **void** reserved word in the declaration of a method, be sure that the method lacks a return statement or the return statement (if used) lacks a value to return. The **void** reserved word prevents returning a value to the calling method.
- If you wish to return a value at the conclusion of a method's execution, use the return reserved word (see the section on the return reserved word).

while (control)

Syntax

while (*boolean-expression*)
    statment;

Description

The **while** loop repeats the statement (or statements, if enclosed in braces) until the boolean expression evaluates to a `false` result. Because of its structure, the statement is not guaranteed to be executed at all. If the expression is `false` the first time it is analyzed, the statement will never be executed. The boolean expression that controls the loop is evaluated before executing the statement portion (body) of the loop.

Example

```
// Loop until alpha is no longer less than the target; increase alpha by two
// each time through the loop.
while (alpha < target)
  alpha += 2;

// Loop while alpha is less than the target and beta is greater than five.
// The body of the loop modifies the values of the alpha and beta variables.
while (alpha < target && beta > 5) {
  alpha++;
  beta = beta - 1;
}
```

Tips

- The statement portion of the **while** loop is not guaranteed to execute at all. Execution depends on the result of the boolean expression.
- If the boolean expression does not resolve to a boolean value (`true` or `false`), a compile-time error is generated.
- To execute multiple statements as the body of the **while** loop, enclose the statements to execute in braces, creating a statement block.
- Be sure that the boolean expression changes as part of the body of the loop. If the value of the expression does not change, the loop will be performed endlessly.
- A **while** loop may contain other loops (do, for **while**), thereby creating *nested loops.*
- Break and continue statements can be used in **while** loops.

Appendix E

The espressou Script

You can get the `espressou` install script from `/home/matt/CSC460/Espresso/espressou` on `java.cs.unlv.edu`. Remember to make the script executable (`chmod 755 espresso`) after you copy it, and also remember to change the username `none` to your own username on `java.cs.unlv.edu` at the top of the file.
The script can be used for a number of tasks:

- Downloading and installing a phase.
- Patching phases if new code gets handed out.
- Checking if a phase is open for handout.
- Installing test files.
- Creating handin files.

Important: Before you start this course determine where you want your code to live. I suggest a directory called `Espresso` in your CS 460 course directory. All the 6 phases will live in directories called `Phase1` though to `Phase6`.

E.1 Downloading and Installing a Phase

Once you have downloaded the `espressou` script you can install a new phase's handout code using the command `./espressou install <phase>`, where `<phase>` is the phase number you want to install (1 though to 6). You cannot install phases that have not been released yet. When you executing this command, you will be prompted for your password on `java.cs.unlv.edu`; this is where the handout code is located, so it is important that you have access to this machine.

E.2 Patching an Existing Install

If the reference compiler is updated it might mean that you need to install a new version of the handout code. The easiest way to do that is to simply download and install the phase again as described in the previous section. When installing a new version the existing version is moved into a directory with the date and time the new files were installed. The command is `./espressou patch <phase> <dir to patch from>`. Depending on the phase number you specify, the appropriate files are copied back into the new installation from the directory you specify.

E.3 Checking if a Phase is Open for Handout

If you want to check if a phase is open for handout you can run the command `./espressou isopen <phase>`.

E.4 Installing the Test Files

You are strongly encouraged to write your own test files, there is quite a large existing set available. If you are working on a machine other than `java.cs.unlv.edu` you might want to use the command `./espressou install tests <dir>` to install the test files in the directory specified in the command.

E.5 Example

Here is an example of installing the utils and phase 1, compiling, patching and unpatching and tarring up the handin file on my Mac Air:

```
[matt@Air:~/Espresso] ./espressou help
Usage:  espressou install <phase number>
        espressou install tests
        espressou patch <phase to patch> <dir to patch from>
        espressou tar <phase>
        espressou isopen <phase>

[matt@Air:~/Espresso] ./espressou isopen 1
matt@java.cs.unlv.edu's password:
Phase 1 is Open

[matt@Air:~/Espresso] ./espressou install tests .
Copying tests files.
matt@java.cs.unlv.edu's password:
Tests.tar                                     100%  610KB 152.5KB/s   00:04
Test files installed in /Users/matt/Espresso/./Tests

[matt@Air:~/Espresso] ll
total 32
drwxr-xr-x   8 matt  staff   272 Aug 20 14:55 Tests
-rwxr-xr-x   1 matt  staff  4293 Aug 20 14:40 espressou

[matt@Air:~/Espresso] ./espressou install 1
matt@java.cs.unlv.edu's password:
Downloading phase 1
matt@java.cs.unlv.edu's password:
Espresso-Phase1-Handout.tar                   100%  640KB 160.0KB/s

Unpacking phase 1 in Phase1/
Done!

[matt@Air:~/Espresso] cd Phase1/
[matt@Air:~/Espresso/Phase1] ls -l
total 40
drwxr-xr-x  13 matt  staff   442 Aug 20 14:02 Include
drwxr-xr-x  12 matt  staff   408 Aug 20 14:02 Lib
```

```
-rw-r--r--   1 matt  staff  2076 Aug 20 14:02 build.xml
-rw-r--r--   1 matt  staff     2 Aug 20 14:02 espresso.info
-rwxr-xr-x   1 matt  staff   101 Aug 20 14:02 espressoc
-rwxr-xr-x   1 matt  staff    86 Aug 20 14:02 espressocr
-rw-r--r--   1 matt  staff   242 Aug 20 14:02 id.txt
drwxr-xr-x  14 matt  staff   476 Aug 20 14:02 src

[matt@Air:~/Espresso/Phase1] ant
Buildfile: /Users/matt/Espresso/Phase1/build.xml

clean:

init:
    [mkdir] Created dir: /Users/matt/Espresso/Phase1/bin

scanner:
     [java] Reading ‘‘src/Scanner/espresso.flex’’
     [java] Constructing NFA : 631 states in NFA
     [java] Converting NFA to DFA :
     [java]
     .....................................................................................
     [java] 159 states before minimization, 122 states in minimized
     DFA
     [java] Writing code to ‘‘src/Scanner/Scanner.java’’

parser:
     [java] Opening files...
     [java] Parsing specification from standard input...
     [java] Error at 131(3): java_cup.runtime.Symbol
     ‘‘compilation_unit’’ has not been declared
     [java] Closing files...
     [java] ------- CUP v0.10j Parser Generation Summary -------
     [java]   1 error and 0 warnings
     [java]   9 terminals, 2 non-terminals, and 2 productions
     declared,
     [java]   producing 0 unique parse states.
     [java]   0 terminals declared but not used.
     [java]   0 non-terminals declared but not used.
     [java]   0 productions never reduced.
     [java]   0 conflicts detected (0 expected).
     [java]   No code produced.
     [java] ---------------------------------------------------
     (v0.10j)

BUILD FAILED
/Users/matt/Espresso/Phase1/build.xml:32: Java returned: 100

Total time: 2 seconds

[matt@Air:~/Espresso/Phase1] cd ..
[matt@Air:~/Espresso] ./espressou tar 1
Buildfile: /Users/matt/Espresso/Phase1/build.xml

clean:
```

```
BUILD SUCCESSFUL
Total time: 0 seconds
Please hand in the 'Espresso-Phase1-matt.tar' file located in the
Phase1 directory without renaming it.
```