Compilers Project 2

Κώστας Χασιαλής - 1115201600195

1 First pass visitor (ClassDefinitions.java)

The first visitor collects every definitions and model them using classes.
For example, a ClassDefintion has a ClassIdentifier and ClassBody, ClassBody has methods and a method consists of a ClassMethodDeclaration and a ClassMethodBody etc.

The first pass visitor catches the following errors :

- Duplicate declarations of class fields
- Duplicate declarations of methods
- Duplicate declarations of classes
- Extends class name not previously declared in file
- Derived-Parent method should prototypes should be identical

2 Second pass visitor (SemanticAnalyzer.java)

The second visitor attempts to (and hopefully it does :D) the rest of the errors.
For example, for an assignment a = b that b is typeof a (or subtype of a).
(NOTE: It also finds errors that "belong" to the first visitor pass (declarations) but first visitor pass is unable to find. For example if we declare a variable A a; and define A later in file)

The way it does this is the following:
Every expression returns an ObjectType.
ObjecType has all the necessary information about this type. More specifically its fields are :

- isPrimitive : boolean (whether or not object is a Custom Object)
- primitiveType : String (the actual type (ex. int))
- identifier : String

- customType : (String, Set<String>) key-value pair. The String of this pair represents the name of the class this type is instance of. For example, if we have A a then customType.String = A.
  The Set<String> of this pair represents the classes that this class extends. For example, suppose we have

```
class D {
    public int foo() {
    }
}

class C extends D {
}

class B extends C {
}

class A  extends B {
}
```

  Then, Set<String> of an instance of A is : A, B, C, D. This means that if we do something like a.foo() then we search if foo is a method that is inside any of the classes of this set. Same happends for fields etc.

The flow of this visitor can be summarized like this : Before entering a method, check if we have a declaration of an instance of one Class. For example A a in class fields of B. If exists such a declaration, check if A is defined in the file. Same goes for method parameters, method fields etc.

Then, for every expression, if this expression is an identifier, tries to find if this identifier is defined, construct its type and return an ObjectType that describe the type of this identifier.
If this expression is not an identifier, it simply constructs the ObjectType and returns it.
By modelizing types with this class (ObjectType) we can simply override "equals" methods and then its very easy to catch any error.
For example, if we have a return expression, there is an error if !returnType.equals(expressionType). Same goes for all statements.

Thus, the hardest part of this visitor is to correctly construct the object type and correctly define the equals methods. The rest of the program is quite trivial.

Finally, one last note.
Assume we have the previous example with the return expression and suppose we have the following code snippet:

```
class A {
}

class B extends A {
    public A getA() {
        return new B();
    }

    public B getB() {
        return A();
    }
}
```

The second method **must** return "cannot convert A to B on return expression" error, but the first method should not. This means that if we do **expressionReturnType.equals(expressionReturnType)** instead of **returnType.equals(expressionReturnType)** then we accept the second method instead of the first one.

This means that we should check if returnType.equals(expressionReturnType) or if expressionReturnType is-a instanceof returnType and not the opposite. Same goes for assignments, method parameters, etc.

I may have ommited some (maybe important) details but I believe the code is well-documented.