Structure of IF clause in Java and meaning of variables in Java and Haskell

Martin Kozeny
CSCI 4501: Programming Language Structure
Spring 2011 University of New Orleans

February 16, 2011

1 Structure of IF clause

The main question is, to which `if` is connected the only one `else` in the piece of code shown below.

```
package main;

public class Main {

  /**
   * @param args
   */
  public static void main(String[] args) {
    ...;
    if(be1)
      if(be2)
        stm1;
    else
      stm2;
    ...
  }

}
```

According to this part of Java Language Specification in BNF,

```
...
Statement:
        Block
        assert Expression [ : Expression] ;
     if ParExpression Statement [else Statement]
     for ( ForControl ) Statement
     while ParExpression Statement
     do Statement while ParExpression   ;
     try Block ( Catches | [Catches] finally Block )
     switch ParExpression { SwitchBlockStatementGroups }
     synchronized ParExpression Block
     return [Expression] ;
     throw Expression   ;
     break [Identifier]
     continue [Identifier]
        ;
        StatementExpression ;
        Identifier   :   Statement
...
```

we can see, how is the `if` statement implemented. In this rule labeled as `Statement`, after terminal symbol `if` comes nonterminal symbol `ParExpression` and then comes again nonterminal symbol `Statement` followed by `[else Statement]`. Following nonterminal symbol `ParExpression` in Java Language Specification, we can see that this symbol is evaluated as boolean expression in `if` condition and after that is recursively called nonterminal `Statement`. In presented source code, nonterminal statement is replaced again by `if` clause and BNF shows, that after `if` clause can be zero or one occurrences of `else Statement` clause (`[else Statement]`). According to this rule, `else` clause is paired with nearest `if` statement so in source code shown above, `else` is paired with `if(be2)` statement. I have verified that also via debugging in Eclipse.

2 Meaning of variables in Java

In Java, we have this kinds of variables according to their scope:

- member variable which could be class or instance variable
- local variable
- method parameter
- exception handler parameter

Member variable A member variable is a member of a class (class variable) or a member of an object instantiated from that class (instance variable). It must be declared within a class, but not within the body of a method of the class. Member variables can also be initialized when they are declared. There is a difference between instance and class variable. Class member is declared with static modifier. The main difference between the class variable and Instance variable is that the first time, when class is loaded in to memory, then only memory is allocated for all class variables. That means, class variables not depends on the Objets of that classes. Whatever number of objects are there, only one copy is created at the time of class loding. The main advantage of this class variable is, suppose if we want to maintain a counter to count the no of object of type that class, in that case we use variable counter as the static variable.

Local variable A local variable is a variable declared within the body of a method or within a block of code contained within the body of a method. Local variables are declared inside the body of the methods or within a block of code (=code enclosed with brackets) contained within the body of a method. The scope of a local variable extends from the point at which it is declared to the end of the block of code in which it is declared.

Method parameters Method parameters are the formal arguments of a method. (A method is a function defined inside of a class.) Method parameters are used to pass values into and out of methods. The scope of a method parameter is the entire method for which it is a parameter. Method parameters are initialized by the values passed to the method or exception handler by the calling program. One great myth is that in methods parameters objects are passed by reference, primitives are passed by value. However in Java is everything passed by value. Objects, however, are never passed at all. The values of variables are always primitives or references, never objects. Let me show small example on the next page.

```
class Test {
  public static void main (String[] args) {
    Person p1 = new Person("Martin Kozeny");
    System.out.println("Person name is " + p1.getName());

    // now let's change name
    changeName(p1);

    /*
     * OK by now, if Java is passing 'p1' as a reference everything
     * in the method 'changeName' shall be reflected.
     */
    System.out.println("Person name NOW is " + p1.getName());
  }

  static void changeName (Person p) {
    p.setName("Michael Schumacher");

    /*
     * If "p" is a reference, then the instance of "p" will now be changed!
     * Else the instance will be the same and name will be "Michael Schumacher"
     */
    p = new Person("Niki Lauda");
  }
}

class Person {
  String _name;

  public String getName() {
    return this._name;
  }

  public void setName(String name) {
    System.out.println("Setting Person name to " + name);
    this._name = name;
  }

  public Person(String name) {
    this._name = name;
  }
}
```

And here is the output of code shown above:

Person name is Martin Kozeny
Setting Person name to Michael Schumacher
Person name NOW is Michael Schumacher

Exception handler parameters Exception handler parameters are arguments to exception handlers. The exception handling mechanism for Java is considerably different from that for C++. Exception handler parameters are initialized by the values passed to the method or exception handler by the calling program.

Source code below is short example of using variables in Java.

```
package var;

public class Variables {

  /**
   * @param args
   */
  /*declare and initialize instance variable*/
  public int instanceVariable=12;

  /*declare and initialize class variable*/
  public static int classVariable=13;

  public static void main(String[] args) {
    System.out.println("Class variable: "
                + classVariable);

    /*Instantiate object to access instance variable and method.*/
    Variables var1 = new Variables();
    System.out.println("Instance variable: "
             + var1.instanceVariable);
    /*method call*/
    var1.method(14);

    /*you can see that class variable is still same*/
    Variables var2 = new Variables();
    System.out.println("Var1 class variable: "
                + var1.classVariable);
    System.out.println("Var2 class variable: "
                + var2.classVariable);

    /*if one of the object change class variable
     * this change is also reflected to other objects,
     * because class variable is 'object independent'*/
    var2.classVariable=16;
    System.out.println("Var1 class variable: "
                + var1.classVariable);
    System.out.println("Var2 class variable: "
                + var2.classVariable);


    /*if one of the object change instance variable
     * this change is not reflected to other objects,
     * because instance variable is connected to concrete object,
     * we can say it is his state*/
    var2.instanceVariable=17;
    System.out.println("Var1 instance variable: "
                + var1.instanceVariable);
    System.out.println("Var2 instance variable: "
                + var2.instanceVariable);


    /*declare and intitialize a local variable*/
    int localVariable = 15;
    System.out.println("Local variable: "
                    + localVariable);

  }
  public void method(int methodParameter)
  {
    System.out.println("Method parameter: "
                + methodParameter);
  }
}
```

And here is the output of the code above:

```
Class variable: 13
Instance variable: 12
Method parameter: 14
Var1 class variable: 13
Var2 class variable: 13
Var1 class variable: 16
Var2 class variable: 16
Var1 instance variable: 12
Var2 instance variable: 17
Local variable: 15
```

3 Meaning of variables in Haskell

Haskell has variables which do not hold the value and cannot be modified - in Haskell there are **immutable variables** or we can say it is **single-assignment language** - it does not allow the reassignment of variables within a scope. For this fact some people say that Haskell does not have any variables but **definitions**, because variable is a binding of a name to a value that varies over a range. In general in Haskell everything is a function. Even if it looks like a 'variable', it is just a nullary function taking no arguments. Assigning some variable does not mean that the 'variable' is calculated. If this value is never needed it will be never calculated, beacause Haskell uses **lazy initialization**. The scope of so called 'variables' can be controlled by using **let** or **where** clauses. **Where** bindings are a syntactic construct that let you bind to variables at the end of a function and the whole function can see them, including all the guards. **Let** bindings let you bind to variables anywhere and are expressions themselves, but are very local, so they do not span across guards.

Haskell has also **type variables**. Type variables nothing similar with variables which we know from Java, C++, ... These variables are used for defining functions and they can be of any type. It allows us to easily write very general functions, also called as **polymorphic functions**.

In conclusion, we can say that there are no 'real variables' in Haskell. Some examples of using so called 'variables' are shown below.

```
-- example of defining variable in 'where' binding
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
     | bmi <= skinny = "You're underweight, you emo, you!"
     | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"
     | bmi <= fat    = "You're fat! Lose some weight, fatty!"
     | otherwise     = "You're a whale, congratulations!"
     where bmi = weight / height ^ 2
           skinny = 18.5
           normal = 25.0
           fat = 30.0

-- example of defining variable in 'let' binding
areaTriangle' :: Float -> Float -> Float -> Float
areaTriangle' a b c
     |(a + b <= c) || (a + c <= b) || (b + c <= a) = error "Sides a, b, c don't describe a
         triangle"
     |otherwise = let s=(a+b+c)/2
     in  sqrt(s*(s-a)*(s-b)*(s-c))

-- example of type variables
head :: [a] -> a
```