A RESOLVE Primer

Svetlana V. Drachova-Strang

Special thanks to my advisor Murali Sitaraman,
and my editors Dr. Krone and A. Isom

Copyright 2010, all rights reserved.

Chapter 1: Introduction to RESOLVE

At-A-Glance

- What is RESOLVE,
- Simple program in RESOLVE
- Facility
- File Inclusions
- Main program
- Assignment Sign
- Comments
- Printing Output on the Screen
- Running RESOLVE
- File types, directory structure, and file names
- Data types
- Operations, modes and return values
- Swap operator

1.0. Hello Resolve!

So, what is RESOLVE? RESOLVE is a unique integrated programming environment, that includes programming language (also called RESOLVE), built-in specification language, compiler, verifier, prover, and in the near future a tutorial. The programming language itself is an object-based language, very similar to object-oriented languages, except it does not handle concurrency, does not implement inheritance, and does not have built-in pointers. It has clean semantics and simple syntax which are easy to learn. You will write a program in RESOLVE, then compile it, which in fact simply translates it into Java code. The Java code is then compiled using a regular Java compiler and the files are executed the same way any Java class files are.

A very interesting and indispensable feature of RESOLVE is the built-in specification language that uses the universal language of mathematical notation. Other modern programming languages can also be specified using specification languages available today, (for example Java and JSP, C# and Spec#), but unlike most of them RESOLVE is fully specified using it's native specification language. And the fact that it uses the language of mathematical notation makes it less ambiguous to write and understand these specifications. The verifier is also an important component of the RESOLVE environment, as fully verified code significantly improves software reliability and decreases failures. The Software Engineering industry is very interested in software verification, but as of today RESOLVE is the only such environment. The verifier and prover are still under development, and a lot of work is still being done by researchers. If you decide you would like to contribute, there is always space for improvement.

Among a variety of modern programming tools available for software engineers, there are other programming tools with built-in specification languages, and there are other object-based languages, but RESOLVE is the only comprehensive environment that offers all of the described above features all-in-one. RESOLVE is an ongoing project that has been partially funded by NASA, NSF, and DoE grants. It has a large number of dedicated

contributors – graduate and undergraduate students, researchers, and professors from several great universities. And because RESOLVE is an ever-evolving entity, you can become one of the contributors as well.

RESOLVE can be easily installed and used on any platform – Windows or Linux, since it's underlying component is a platform independent Java interpreter. The best way of working with RESOLVE is using Eclipse with a special plug-in, also available for both operating systems. RESOLVE has a homepage that contains a lot of useful information, documentation, and installation instructions: http://www.cs.clemson.edu/~resolve/compiler-verifier.html.

Because RESOLVE is so new, there are very few examples available for students to learn syntax and simple concepts, and currently there are no textbooks written on the subject. This manual is intended to get you started with the programming language itself by simply introducing you to some language specifics and syntax. It will not teach you any complex software engineering concepts, or make you into a good programmer. This manual discusses topics that will be most helpful, and provides examples to illustrate the material. And to get it all rolling, you will now see a small program written in RESOLVE.

1.1. A Simple Program in RESOLVE

Like many other books using the cliché program that prints out "Hello World!", the simple program below prints the "Hello RESOLVE!" statement.

Hello_Resolve.fa

```
1.  Facility Hello_Resolve;
2.      uses Std_Char_Str_Fac;

3.      Operation Main();
4.         Procedure
5.            Var hello:    Char_Str;
6.            Var resolve:  Char_Str;

7.            hello   := "Hello";
8.            resolve := "RESOLVE";

9.            -- print "Hello RESOLVE" using two character strings
10.           Write(hello);
11.           Write("  ");
12.           Write_Line(resolve);

13.    end Main;
14.    end Hello_Resolve;
```

When we run the program the output will look like this:

```
Hello Resolve!
```

The rest of the chapter explains in detail what this code does and introduces some concepts that you need to know to successfully write you own program in RESOLVE.

1.2. RESOLVE Program structure

1.2.1. Facility

Line 1 declares a facility. This is similar to Java, where every file is in fact a class. And like Java, facility's name should be the same as the name of the file that declares it. The file extension is ".fa", indicates this is a facility. There are several other types of files in RESOLVE that will be discussed later, but for now you just have to remember to add ".fa" to your facility file name. So, if you created a facility called "Hello_Resolve", then your file name should be named **Hello_Resolve.fa**. The underlying reason for this similarity to Java is that RESOLVE code gets translated to Java, and Java requires that the file name matches the class name. To end the facility, you simply need an **end Hello_Resolve;** statement, as shown in line 14. And, needless to say, all statements end with a semicolon, like so many other programming languages.

1.2.2. File Inclusions

Line 2 of the program states that the program uses **Std_Char_Str_Fac**, which is the built-in facility that allows RESOLVE to handle a specific data type, a character string in this case. This facility contains the **Char_Str** (character string) definition and operations that you can perform on the strings. This is somewhat similar to other programming languages where you have to include header files. You will see more of this later.

1.2.3. Operation Main

Line 3 declares Operation **Main**. This is the main program that does all the work for you. Line 13 contains the statement indicating the end of **Main**. Operation **Main** has a procedure that contains variable declarations and statements. The procedure does not need a closing **end** statement. The procedure declares several variables of type **Char_Str** – character string. The syntax for variable declaration is simple:

```
Var varname:  Vartype;
```

You can either declare one variable per line as we did in this example, or several at once, and you can use keyword "Variable", or it's abbreviated form "Var" as in the example below :

```
Var One, Two: Char_Str;
Variable temp, temp2: Integer;
```

Lines 7 and 8 initialize two variables of type character string. As in Java, RESOLVE strings use double quotes.

1.2.4. Assignment Symbol

An assignment symbol in RESOLVE is a colon followed by an equal sign: **:=** . An equal sign alone, as in mathematics, symbolizes equality, not assignment in RESOLVE. You saw the assignment operator on lines 7 and 8 where character variables are initialized. Similarly, if you saw **my_int := 5;** this is an initialization, and if you see **my_int = 5;** this is a check for equality. Unlike in C, C++, or Java double equal sign "**==**" does not exist in RESOLVE.

1.2.5. Comments

Line 9 of the program is a comment, as indicated by a double dashed (- -) line. This is a single line comment. If you have several lines of comments you can do one of the two things: either make several lines of single-line comments, or use a multi-line comment. A multi-line comment starts with **(*** and ends with ***)** . The two ways are illustrated below:

Example of singe- and double-line comments

```
   -- This is one way to write a very long comment
   -- that spans several lines
```

```
 (* Or you can use this multi-line comment which is
     similar to the C-style comment and can span
     several lines   *)
```

1.2.6. Printing Output

RESOLVE has two operations for printing output to the standard output stream: **Write(..)**, and **Write_Line(..)**. The only difference is that the second one also prints a newline character. You will see more of this later, but every simple built-in type in RESOLVE (integers, booleans, characters, and character strings) has these operations. It is important to remember that these operations expect either a literal or a variable of the correct type as a parameter. If you leave the parenthesis empty the compiler will generate an error.

1.3. Running Resolve code

Running a program in RESOLVE, as with other languages, depends on what user interface is being used. In this section we talk about a command line interface. But if you are using Eclipse (which is recommended) you will have to do some initial setup. Help setting up your running environment is available on the RESOLVE homepage. In any case, your RESOLVE program will go through several stages. After you have typed and saved the above facility file in the correct directory (directories are discussed later) you will need to run the RESOLVE compiler. As mentioned above, your file will have the same name as the facility that you declared in that file, **Hello_Resolve.fa** in this case. The first step is translating your code to Java:

```
java edu.clemson.cs.r2jt.Main -translate +bodies Hello_Resolve.fa
```

The above command translates RESOLVE to Java and generates Java code bodies, just as the flags in the above command indicate. You will see compilation information scrolling down the screen, hopefully without generating any errors. As the result of this command the file **HelloResolve.java** appears in the same directory. From now on this file is treated as a regular java file, and java compiler is used to compile and run the code:

```
java Hello_Resolve.java
javac Hello_Resolve
```

At this point the output prints on the screen, and you are done.

1.3.1. Types of Files in Resolve

There are several types of files in RESOLVE. The example above is a facility file. As mentioned earlier, facilities are executable programs, similar to main programs. They reside in the corresponding directory and have an extension ".fa".

Another type of file is a concept. If we were to draw an analogy between RESOLVE and some object-oriented language, for example C++, then we can say that the concept is similar to a header file, or class declaration. A RESOLVE concept file contains the data structure template that includes the mathematical model used to represent that type (eg.: a string, a set, or a natural number), the constraints (eg.:the min and max ranges of integers), and a list of all operations along with their pre- and post-conditions. There are two important keywords used to specify pre- and post-conditions of RESOLVE operations: **requires** and **ensures**. The **requires** clause specifies the operation pre-condition, while **ensures** is the post-condition. As in other languages, if the pre-condition is met, the post-condition will be met as well, provided the operation is implemented correctly. RESOLVE concept files have an extension ".co" and live in the Concepts directory as shown on the chart below. For example, if we have a Stack template, it's concept file indicates that stack is modeled using a mathematical String of entries, and has operations **Push**, **Pop**, **Depth**, **Rem_Capacity**, and **Clear.** Data structure templates usually contain a list of orthogonal operations, the minimum necessary to ensure the proper functionality of that data structure. For example, stack does not contain the Reverse operation. If we wanted to reverse a stack, we could achieve this by using a combination of Pushes, and Pops on a temporary stack.

Concept files do not reveal how template operations are implemented. This is achieved in a RESOLVE realization – implementation file. As in other languages, every concept can have more than one realization, depending on what the developer needs are at that time. For example, we can model a stack data structure using a mathematical String, and use either an array, a reversed array, or a linked list to implement it. Realizations have an extension ".rb", and are located in the same directory as the concept files.

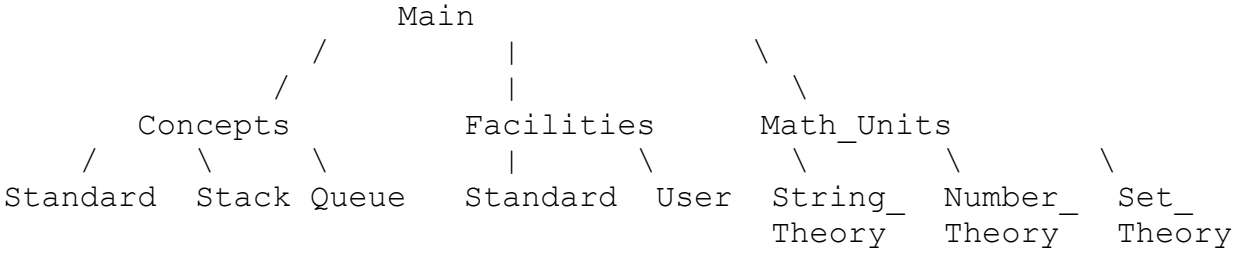
Another type of file is Enhancement. As the name implies, enhancements are used to provide some special functionality to a data structure; that is, enhance it. For example, the stack template does not contain an operation to reverse a stack, so if we wanted to

reverse the stack, we could either use a combination of pushes and pops on a temporary stack in our facility, the possibility mentioned above, or we can write an enhancement that does just that, and use it without having to write a lot of extra code. Plus, enhancements are reusable modules, and component reuse is extremely important in software engineering. Enhancement itself is merely a declaration of a custom operation and it's specifications (requires and ensures clauses). Enhancements are realized by enhancement realizations, the same way concepts are realized by concept realizations. And similar to concept realizations, they can have more than one implementation. Enhancements have an extention ".en", and their realizations have extension ".rb". Both live in the same directory as concepts and concept realizations. To make sure we can tell by looking at the list of files whether a certain file is a concept realization or an enhancement realization, you should use descriptive filenames.

1.3.2. Directory Structure

Below is a chart showing Resolve hierarchy. There are several types of files in RESOLVE – concepts, realizations, enhancements, and facilities, and they live in certain subfolders in the root directory **resolve\workspace\RESOLVE\Main**. Depending on their type, all files need to be placed into the correct location, otherwise the compiler will not be able to find them. The root directory Main contains three subdirectories: **Concepts, Facilities,** and **Math_Units**.

Nice diagram is coming soon



When you download the workspace archive from RESOLVE homepage, take a look at the directory structure. **Main/Math_Units** contains a number of files for the different mathematical theories used in RESOLVE – **String_Theory** , **Number_Theory**, **Set_Theory**, and others, which, at the moment, are syntactical placeholders. **Facilities/Standard** contains facilities for simple built-in types – integers, characters, booleans, etc. **Facilities/User** has folders for different user-created complex types. **Concepts/Standard** further contains folders for each of simple built-in types: integer, character, boolean, etc. Each subfolder contains concept files, enhancements, and realizations. Other folders inside the Concepts directory are for complex built-in data types - **Basic_Map**, **One_Way_List**, and etc. It is important that the files go to the appropriate directory. Browse your workspace and make sure to familiarize yourself with it's structure.

1.3.3. Filenames

As mentioned above, one of the rules in RESOLVE requires that all files have the

same name as the corresponding entity they declare. For example, if you are declaring a concept Stack_Template with a corresponding realization called **Simple_Stack_Realiz**, **Reversal_Capability_Enh** enhancement, and **Mystery_Reversal_Enh_Realiz** then your files will be named: **Stack_Template.co**, **Simple_Stack_Realiz.rb**, **Reversal_Capability_Enh.en**, and **Mystery_Reversal_Enh_Realiz.rb**. If this rule is not followed, errors will be generated during compilation.

1.4. Data Types in Resolve

RESOLVE has two kinds of data types: simple and complex. Simple data types are integers, booleans, characters and character strings. Complex data types are built-in data structures, such as stacks, linked lists, trees, etc. Both built-in simple and complex types are part of the RESOLVE, and live in the **Standard** directory. Users can create their own complex types which will be placed in the **User** directory. Both simple and complex data types have a concept file with a mathematical model, constraints, and a list of operations available for that type. As mentioned earlier, template files have at least one corresponding realization file. This is true for all the built-in complex types, but not for the simple types. Integers, characters, strings, and booleans do not have a RESOLVE realization file. Because RESOLVE files get translated to Java files, and because these simple types directly correspond to the Java integers, characters, strings, and booleans, we use Java to implement them, and skip the step of creating a concept realization file in RESOLVE. Instead, their facility file specifies that they are directly implemented in Java.

1.5. Operations, Parameter modes, and Return types

Before discussing simple data types, you will be shown the syntax of a typical operation. The keyword **Operation** is capitalized, but you can you an abbreviated version of it: **Oper**. The name of the operation below is **Is_Zero**, and the parenthesis includes parameters along with the mode.

```
Operation Is_Zero(evaluates i: Integer): Boolean;
        ensures   Is_Zero = ( i = 0 );
```

The list of parameters is followed by a colon, return type and a semicolon to terminate the statement. Notice how the return value here has the name **Is_Zero** which is the same as the name of the operation itself. An equal sign "=" is a check for equality, not an assignment operator which is ":=". The keyword **ensures** is a post-condition indicating that after the operation terminates the return value will be true if the integer is equal to zero.

In another example, operation does not return any values; the pre-condition here is that the incremented integer i does not exceed the value of the max_integer. Post-condition ensures that the new value of i is the previous value of i incremented by 1. The parameter mode "reassigns" indicates that the value of i is now reassigned, or changed to the newly incremented value. The incoming, or previous value is indicated by a "#" sign. So the **ensures** clause says that the new value of i is the incoming value of i incremented by 1.

```
Operation Increment(reassigns i: Integer);
        requires i + 1 <= max_int;
        ensures   i = #i + 1;
```

1.5.1. Parameter Modes

There are several parameter modes in RESOLVE. They are passed along with the variable name and type to the operations to indicate what happens to the incoming variable after the operation is complete.

evaluates – the variable is evaluated for a specific property and remains unchanged.
preserves – the value of the incoming value will be preserved.
alters – the variable will be changed in an unspecified way.
restores – after certain manipulations, the variable value will be restored to it's incoming value.
replaces – value will be replaced by some other variable.
updates – the value will be changed in an unspecified way.
clears – the variable will be reset to the initial value as defined in it's template. For example, Booleans will be set to true, and Integers to zero.

Parameter modes will be discussed in more detail in Chapter 3, but you will see some examples on how to use parameter values below.

```
Operation Addition ( preserves int1: Integer, preserves int2: Integer,
                     replaces sum:Integer);
```

Operation Addition takes three parameters: two integers that will be added together, int1 and int2, and sum that will contain the sum of the previous two. After the operation completes the values of int1 and int2 will be the same as before the operation was called, because both of them are passed with the **preserves** mode. The value that the variable sum had at the time operation was called will be replaced by the sum of int1 and int2, as indicated by it's mode **replaces**.

1.5.2. Return Values

RESOLVE operations can also return values. The two examples below show the correct syntax of an operation returning a value:

```
Operation Some_Op(alters c1: Char, clears c2: Char): Integer;
Operation Mystery_Op(restores Data: Integer, updates S: Integer) : Boolean;
```

Operation **Some_Op** takes two parameters and returns an Integer value, and operation

Mystery_Op returns a Boolean value.

1.6. Swap Operator

RESOLVE has an operator that swaps the values of two variables without the use of the temporary variable. The swap operator can be used with any type, even the complex user-created data types. The only rule is that the two values being swapped must be of the same type. For example, we can swap two integers:

```
Var i, j : Integer;
i := 1;
j := 3;
i :=: j;
```

After we execute the last statement, the values of i and j are now 3 and 1, respectively. Notice we did not have to declare and use a temporary variable; RESOLVE does it for us behind the scenes. We could use the same operator to swap two characters, strings, stacks or linked lists. None of the template files declare the swap operation. This unique feature is part of the RESOLVE itself, and all you need to do to use it is use the swap operator.

Chapter2: Simple Data Types

At-A-Glance

- Integers
- Characters
- Booleans
- Character Strings

2.0. Simple Data Types

Simple RESOLVE data types are integers, characters, booleans, and character strings. At this time RESOLVE does not have built-in floating point numbers, but you can be the first to implement them. The simple types have corresponding concept files in the directory **Main/Concepts/Standard**, but do not have corresponding concept realization files written in RESOLVE . Instead, because they directly correspond to Java simple built-in types, they are implemented in Java, as specified in their facility files. These simple data types are discussed in detail below.

2.1 Integers

The Integer template file is located in **Main/Concepts/Standard/Integer** directory. If you examine it, you see that RESOLVE integers are modeled by a natural number. The initialization clause specifies that a new instance of an integer will be initialized to zero when it is created, and that it's value will be constrained to the range of less than max_int and more than min_int.

```
Type Family Integer is modeled by Z;
        exemplar i;
        constraint min_int < i < max_int;
        initialization ensures i = 0;
```

Integer template defines a variety of operations which are shown in the table below, the names of which explain unambiguously what they do. Only a couple of these operations need to be mentioned, mostly to explain their syntax, which is shown in the table below.

```
Operation Is_Zero(evaluates i: Integer): Boolean;
Operation Is_Not_Zero(evaluates i: Integer): Boolean;
Operation Increment(reassigns i: Integer);
Operation Decrement(reassigns i: Integer);
Operation Less_Or_Equal(evaluates i, j: Integer): Boolean;
Operation Less(evaluates i, j: Integer): Boolean;
Operation Greater(evaluates i, j: Integer): Boolean;
Operation Greater_Or_Equal(evaluates i, j: Integer): Boolean;
Operation Sum(evaluates i, j: Integer): Integer;
Operation Negate(evaluates i: Integer): Integer;
Operation Difference(evaluates i, j: Integer): Integer;
Operation Product(evaluates i, j: Integer): Integer;
Operation Power(evaluates i, j: Integer): Integer;
```

```
Operation Divide(evaluates i, j: Integer; replaces q: Integer);
Operation Mod(evaluates i, j: Integer): Integer;
Operation Rem(evaluates i, j: Integer): Integer;
Operation Quotient(evaluates i, j: Integer): Integer;
Operation Div(evaluates i, j: Integer): Integer;
Operation Are_Equal(evaluates i, j: Integer): Boolean;
Operation Are_Not_Equal(evaluates i, j: Integer): Boolean;
Operation Replica(restores i: Integer): Integer;
Operation Read(replaces i: Integer);
Operation Write(evaluates i: Integer);
Operation Write_Line(evaluates i: Integer);
```

Integer template defines operation **Read()** that takes input from a standard input stream (keyboard) and replaces the integer that was passed to it with this newly provided value. **Operations Write()** and **Write_Line()** output to the standard output stream (screen) the value of I, and the only difference is that the second also prints a newline. Operation **Replica()** makes a copy of the integer i that is passed to it and then returns that copy. The value of i remains unchanged. All other operations need no explanation.

Some of the RESOLVE operators are different from what you have seen in other programming languages, as shown below.

```
:=    assignment
 =    check for equality
/=    not equal
**    power
:=:   swap
```

When the user writes a program that uses RESOLVE Integers, the **Std_Integer_Fac** facility file should be included at the top using statement **uses Std_Integer_Fac;**. This is similar to including a header file in C++ code to tell the program where to look for the definition of the type. The **Std_Integer_Fac** file contains only a few lines of code (shown below) and indicates that integer template is realized by **Std_Integer_Realiz** in Java.

```
Facility Std_Integer_Fac
    is Integer_Template
    realized by Std_Integer_Realiz;
```

The integer template file also includes **Std_Boolean_Fac** file. This enables us to return Boolean values from operations, as well as **Integer_Theory** which contains mathematical definitions of integer operators. Forgetting to include these important files will generate compilation errors.

Example of a program using RESOLVE Integers

```
Facility Integer_Test
          uses Std_Integer_Fac,  Std_Boolean_Fac, Std_Char_Str_Fac;

     Operation Main();
         Procedure
             Var input1, input2:  Integer;
             Var sum, power, modulus:  Integer;

             -- ask user to input two integers
             Write ("Please enter two integers: " );
             Read (input1);
             Read (input2);

             - calculate sum, power, and modulus
             sum := Sum (input1, input2);
             power := Power (input1, input2);
             modulus := Mod (input1, input2);

            - print results out
             Write (" The sum is:" );
             Write_Line (sum);

             Write (" The power is:" );
             Write_Line (power);

             Write (" The modulus is:" );
             Write_Line (modulus);
             Write_Line (" ");

       end Main;
 end Integer_Test;
```

2.2. Characters

The Character template file is located in **Main/Concepts/Standard/Character** directory. If you examine it, you will see that RESOLVE characters are modeled by a natural number. It is very simple: what we see is a character; while the compiler sees a number. The initialization clause specifies that an instance of a character will be initialized to zero when it is created, and that the constraint is that it's value will be less than the Last_Char_Num, which is defined below as a natural number greater than zero.

```
Definition Last_Char_Num: N;
    Constraint Last_Char_Num > 0;

    Type Family Character is modeled by N;
        exemplar c;
            constraints c < Last_Char_Num;
            initialization
                  ensures c = 0;
```

The list of Character operations is given in the table below:

```
    Operation Char_to_Int(evaluates c: Character): Integer;
    Operation Are_Equal(evaluates c1, c2: Character): Boolean;
    Operation Are_Not_Equal(evaluates c1, c2: Character): Boolean;
    Operation Less_Or_Equal(evaluates c1, c2: Character): Boolean;
    Operation Less(evaluates c1, c2: Character): Boolean;
    Operation Greater_Or_Equal(evaluates c1, c2: Character): Boolean;
    Operation Greater(evaluates c1, c2: Character): Boolean;
    Operation Replica(restores c: Character): Character;
    Operation Read(replaces c: Character);
    Operation Write(evaluates c: Character);
    Operation Write_Line(evaluates c: Character);
```

These operations hardly need to be explained. Remember to include **Std_Character_Fac** in your programs to tell the compiler where to look for the type. Similar to the **Std_Integer_Fac**, **Std_Character_Fac** facility declares that Character is modeled in the Character template and is realized by **Std_Character_Realiz** in Java.

```
Facility Std_Character_Fac
    is Character_Template
    realized by Std_Character_Realiz;
```

Example of a program using RESOLVE Characters

```
-- Example of character read/write
Facility Character_RW_Example;
       uses Std_Boolean_Fac, Std_Character_Fac, Std_Integer_Fac;

    Operation Main();
    Procedure
      Var C, Dollar: Character;

      Dollar := '$';
      Read(C);
      While (C /= Dollar)
            changing C;
            maintaining true;
            decreasing 0;
      do

            Write(Char_to_Int(C));
            Read(C);
      end;
    end Main;
```

```
end Character_RW_Example;
```

While/Do loop

In the above example you also see how to use a **While/do** loop. Here the user enters a value of C, which is compared to the dollar sign as the condition to the While clause. "changing C" means that we will be changing the value of this variable within the loop as we read another value into C in the "do" clause. The "maintaining true" means that we are maintaining the true condition of the while clause. You will see more on while loops in Chapter 3.

2.3. Booleans

The Boolean template is located in **Main/Concepts/Standard/Boolean** directory. An instance of a Boolean variable is initialized to true.

```
    Type Family Boolean is modeled by B;
        exemplar b;
        initialization ensures b = true;
```

A list of Boolean operations is given below.

```
    Operation True(): Boolean;
    Operation False(): Boolean;
    Operation And(evaluates b1, b2: Boolean): Boolean;
    Operation Or(evaluates b1, b2: Boolean): Boolean;
    Operation Not(evaluates b: Boolean): Boolean;
    Operation Are_Equal(evaluates b1, b2: Boolean): Boolean;
    Operation Are_Not_Equal(evaluates b1, b2: Boolean): Boolean;
    Operation Replica(restores b: Boolean): Boolean;
    Operation Read(replaces b: Boolean);
    Operation Write(evaluates b: Boolean);
    Operation Write_Line(evaluates b: Boolean);
```

As you have already noticed, the last four operations are common to all the simple types. The other operations used in the boolean template are **True** and **False**, logical operations **And**, **Or**, **Not**, and comparisons **Are_Equal** and **Are_Not_Equal. Std_Boolean_Fac** file is similar to the one in the Integer directory.

```
Facility Std_Boolean_Fac
    is Boolean_Template
    realized by Std_Boolean_Realiz;
```

Example of a program using RESOLVE Booleans

```
-- Example with Booleans
```

```
Facility Boolean_Test;
       uses Std_Boolean_Fac, Std_Character_Fac, Std_Integer_Fac;

    Operation Main();
    Procedure
      Var result, : Boolean;
      Var int1, int2: Integer;
      Var ch1, ch2: Character;
      ch1 := 'a';
      ch2 := 'b';
      int1 := 1;
      int2 := 2;

      result := Are_Equal(ch1, ch2);

      If (result = True())
      Then Write_Line(" Characters are the same");
      Else Write_Line(" Characters are NOT the same");
      end;

      result := Greater_Or_Equal(int1, int2);
      If (result = True())
      Then Write_Line(" Integer1 is greater or equal to Integer2");
      Else If (result = False())
      Then Write_Line(" Integer1 is less than Integer2");
      end;

    end Main;

end Boolean_Test;
```

In the above example you not only saw how to use Booleans, but also how to write a simple **If/Then/Else** statement.

2.4. Character Strings

The template file for character strings is found in **/Main/Concepts/Standard/Char_Str** directory. Character strings are modeled using a mathematical string of length N, as shown below. Newly created strings are initialized to an empty string, and the length of a string cannot be more than the specified maximum length.

```
Definition Max_Char_Str_Len: N;

    Type Family Char_Str is modeled by Str(N);
        exemplar s;
        constraints |s| <= Max_Char_Str_Len;
        initialization
            ensures s = empty_string;
```

A list of Char_Str operations is shown below:

```
Operation Char_Str_for(evaluates c: Character): Char_Str;
Operation Are_Equal(evaluates s1, s2: Char_Str): Boolean;
Operation Are_Not_Equal(evaluates s1, s2: Char_Str): Boolean;
Operation Merger(evaluates s1, s2: Char_Str): Char_Str;
Operation Length(evaluates s: Char_Str): Integer;
Operation Replica(restores s: Char_Str): Char_Str;
Operation Read(replaces c: Char_Str);
Operation Write(evaluates c: Char_Str);
Operation Write_Line(evaluates c: Char_Str);
```

The last four operation are common to those of other simple type templates. **Operation Char_Str_for( )** converts a character to the corresponding character string. **Length( )** returns the length of the String, and **Merger( )** takes two strings and merges them in one.

Just like other simple types, the concept realization is implemented directly in Java. You already saw a simple example of a program using character strings in Chapter 1.

Chapter 3: Writing conditional statements, loops, procedures and facilities – basic elements of a RESOLVE program.

At-A-Glance

- If-Then_Else Statement
- While loop
- Keyword Abbreviations
- Procedures and Operations
- Facilities and Facilities with Parameters

This chapter discusses such elements of a simple RESOLVE program as conditional statements, loops, keyword abbreviations, explains the difference between operations and procedures, and explains how to write facilities, including facilities that have parameters. You will see some examples.

3.1. If-Then-Else statements

The basic syntax of an If-Then-Else statement is given below:

```
If   (condition)   then
     statement_sequence;
end;
```

It also has a longer form:

```
If    (condition)  then
      statement_sequence;
elseif   (condition)  then
      statement_sequence;
else    statement_sequence;
end;
```

Now we will look at a small example that declares three variables, compares them, and prints some messages on the screen.

Example 1: simple If-Then-Else

```
Facility Example1;
       uses Std_Integer_Fac, Std_Char_Str_Fac;

    Operation Main();
    Procedure
       Var temp, temp2: Integer;
       temp  := 5;
       temp2 := 6;
```

```
      If (temp > temp2)  then
              Write("temp is greater than temp2");
          else Write("temp2 is greater");
      end;

    end Main;
end Example1;
```

Example2:

```
Facility Example2;
       uses Std_Integer_Fac, Std_Char_Str_Fac;

    Operation Main();
    Procedure
      Var temp: Integer;

      temp := 5;

      If (temp = 1) then Write(1);
      -- elsif (temp = 2) then Write_Line(2);
      else Write_Line(3);
      end;

    end Main;

end Example2;
```

Example 3: nested If-Then-Else statements:

```
Facility Example3;
       uses Std_Integer_Fac, Std_Char_Str_Fac;

    Operation Main();
    Procedure
       Var debug, temp, temp2: Integer;
       temp  := 5;
       temp2 := 6;
       debug := 1;

      If (debug = 1) then
           Write_Line("Debug flag ON");

           If (temp > temp2) then
               Write (temp);
               Write_Line("is greater than");
               Write_Line(temp2);

           else  Write_Line("temp2 is less than temp");
           end;

      else  Write_Line("Debug flag is OFF");
```

```
      end;

    end Main;
end Example3;
```

3.2. In Case you are Feeling Adventurous :)

RESOLVE is a revolutionary environment, and is just becoming popular with programmers and software engineers. As you may have already discovered, examples are not always readily available, as for some older languages such as C, C++, or Java. So, if you are a programmer trying to figure out the correct syntax for a specific programming structure, where would you look? As I found out, if everything else fails, the best way to do is to look at the source. That is, the compiler's source code itself. Specifically, you will need to examine RESOLVE compiler's parser.

Parser is the part of the compiler responsible for parsing code into syntactical units. To examine the parser you would need to download the source code for the RESOLVE compiler itself, and examine the RParser.g file located in the folder called “parsing”. For your class you have only downloaded the workspace with code, and a compiler executable file, but the compiler source code can also be downloaded free from sourceforge, and installation instructions are available on the RESOLVE page. The file contains syntax not only for the If-Then_Else statements, but for all the imaginable RESOLVE entities. In my opinion the best way to look at it is in Eclipse. If you were to examine the Rparser.g file, this is what you would find:

```
if_statement
    :   IF^ condition
        THEN! statement_sequence
        (elsif_item)*
        (else_part)? END!
    ;

elsif_item
    :   ELSIF^ condition
        THEN! statement_sequence
    ;

else_part
    :   ELSE^ statement_sequence
    ;
condition
    :   program_expression
    ;

statement_sequence
    :   (statement SEMICOLON!)*
        { ## = #([STATEMENT_SEQUENCE, "Statement sequence"], ##); }
    ;
```

It is not as hard as it looks. If you are familiar with regular expressions you know that the

asterisk(*) is used to indicate that an item can be used zero or more times, (!) means it has to have exactly one, and (?) that it can have zero or one of this item. This way if you look at the if_statement from the parser file mentioned above you can see that the keyword IF is followed by a condition. This will be followed by keyword THEN and a statement_sequence. And that in it's turn is followed by zero or more elsif items, which is followed by zero or one else parts, and is finished by the keyword END.

3.3. While loop

If you check the same parser file it tells you the syntax for the While loop:

```
while_loop_statement
    :   WHILE^ condition
        (changing_clause)?
        maintaining_clause
        (decreasing_clause)?
        DO! statement_sequence END!
    ;
```

This simply means that the **While** keyword is followed by a condition, then it may or may not have one changing clause, followed my one maintaining clause, followed by zero or one decreasing clause. Finally there is a **do** keyword followed by a sequence of statements, and the keyword **end;** at the end.

```
While (condition)
    changing ... ;
    maintaining ...;
    decreasing ...;
do statements;
end;
```

Example 4 – While loop.

```
Facility Example4;
       uses Std_Integer_Fac;

    Operation Main();
    Procedure
       Var temp, temp2, temp3: Integer;
       temp2 := 2;
       temp3 := temp2;
       temp  := 5;

       While(temp > 0)
         maintaining temp3 = temp2;
         decreasing temp;
       do
         Write_Line(temp3);
         Write_Line(temp2);
         Write_Line(temp);
       end;
```

```
    end Main;
end Example4;
```

In the Example 4 above we maintain the values of temp 2 and temp3, and only decreasing temp. It is important to remember that omitting maintaining clause is syntactic error. Also, if you use keyword increasing instead of decreasing, you will get a compilation error as well, since there is no such a keyword in RESOLVE. However, it is still possible to count up in the loop. To do it you will omit the decreasing clause and just increment your count value in the do clause. Also, if you do not have anything that you are maintaining in the program, and compiler still expects a maintaining clause – use "maintaining true;" as shown below.

```
Facility Example5;
       uses Std_Integer_Fac;
    Operation Main();
    Procedure
       Var temp, temp2, temp3, temp4: Integer;
       temp4 :=0;

       While( temp4 < 5)
         maintaining true;
       do
         temp4 := temp4 + 1;
         Write_Line(temp);
       end;

   end Main;

end Example5;
```

3. 3Difference between Operations and Procedures

3.3.1. Templates or enhancements

When you are writing a new data structure as a template, for example Stack_Template.co, or an enhancement for any template, you will only be declaring Operations along with their requires and ensures clauses. This file does not contain implementations of the operations. Code below shows how you can declare an operation for a Stack tempate.

```
Operation Pop(replaces R: Entry; updates S: Stack);
       requires |S| /= 0;
       ensures #S = <R> o S;
```

3.3.2. Realizations

Implementation for your concept or enhancement will be done in a realization file, so, for example, if you have written an Array_Realization for your Stack, your procedure Pop will look like that:

```
Procedure Pop(replaces R: Entry; updates S: Stack);
```

```
        R :=: S.Contents(S.Top);
        S.Top := S.Top - 1;
    end Pop;
```

Notice here, that you do not use the “end;” to end the operation declaration in your concept or enhancement file, but you use it in the realization file to end the implementation of the Pop. Also, So far this is very straightforward.

3.3.3. Facilities:

When you are writing a Facility, you will use both keywords Operation and Procedure. Keyword “Operation” is used to declare it, and keyword “Procedure” refers to it's implementation. For example in the Examples 1 through 5 of this chapter you saw how to declare Operation Main. Procedure below contains statements that implement the Main. Notice here that you have to end your Operation Main with the “end;” It is that simple.

```
Operation Main();
   Procedure
      Statements;

end Main;
```

3.3.4. Local and helper operations:

Sometimes you will write a local “helper” operation inside your realization or facility files. Such an operation is not a part of the template operation implementation, it is a utility function that will be used by one of the template Operations. In this case it will be very similar to how you declared Operation Main in the example above. For example, if you have a Clean_Array_Realization that implements Stack_Template, you may create a helper operation that will be called by a template operation Clear. In this case you will also need to terminate it with the “end;” statement.

```
Operation Clear_Char(clears C: Character);
   Procedure
     C := ' ';
end Clear_Char;
```

4. Keyword abbreviations and substitutions in RESOLVE

RESOLVE have one very convenient feature. It allows you to abbreviate some of the keywords, or avoid capitalization. Below are the keywords you are most likely to encounter:

Keyword:	**Allowed:**
alters	alt
clears	clr
evaluates	eval

preserves	pres
replaces	rpl
restores	rest
updates	upd
reassigns	rea
Procedure	Proc
Operation	Oper, operation, oper
Variable	Var
Variables	Vars
Constraints	constraints
Constraint	Constraint
Convention	convention
Correspondence	correspondence
Property	Pty
If	if
Definition	Def, definition, def

3.5. Writing Facilities

3.5.1. Facilities with Parameters.

Chapter 4: Understanding Parameter Modes, Ensures and Requires Clauses

Ensures and Requires Clauses

There are several parameter modes in RESOLVE that indicate exactly what happens to the variables passed as parameters to an operation. For example in the following operation

Operation Mystery (clear b1: Boolean, restores b2: Boolean)
    requires
    ensures

… finish example here

First of all, a note about ensures and requires clauses. Requires clause is what we commonly call a precondition to the function, and ensures is a postcondition. It is the responsibility of the caller (program that calls that operation) to ensure that the precondition holds. Generally speaking, requires clause is the requirement that needs to be met before the operation can be called. Assuming the operation is implemented correctly If the precondition is true then the postcondition is true as well, provided the internal mechanism of the operation is correctly implemented.

Parameter Modes

In the above example we know that operation Mystery takes two Boolean parameters b1 and b2, and that after operation is completed

Exercises
Chapter 1

1. Name several components of the unique RESOLVE environment and explain why it is unique
2. Show two different ways of writing comments in RESOLVE.
3. Identify several components of a RESOLVE facility.
4. What is the difference between operators ":=" and "=".
5. Write a line of code printing the phrase "RESOLVE is gr8 !!!!".
6. What is the difference between operations Write(some_string); and Write_Line(some_string);
7. Name the steps involved in getting RESOLVE program to run after you type the code.
8. Name several different types of files in resolve.
9. There are two clauses "ensures" and "requires". Explain each.
10. Match the files and their extensions:

a. facility	1 .co
b. realization	2 .fa
c. template	3 .en
d. enhancement	4 .rb

11. True or False:
    a. Enhancements do not have realizations
    b. Templates show how operations are implemented.
    c. Realizations show how operations are implemented.
    d. Every concept file can have more than one realization.
12. You have a file that declares a facility My_Test_Facility. What is the name of the file that I resides in?
13. Write a line showing a declaration of operation Mystery that has integer Temp as a parameter, and returns a Boolean. Use evaluates mode.
14. You have an operation:
    Operation Mystery_Op(clears str: String, updates i: Integer);
    which increments integer i, and prints the string to the screen: str := "hello" and i := 5; What values will str and I hold after the operation was called.
15. You have the following line of code:

```
Var int1, int2: Integer;
int1 := 11;
int2 := 17;
int1 :=: int2;
```

    What are the values of int1 and int2 now?

Answers

Chapter 1

1. Programming language, built-in specification language, compiler, verifier, prover.
2. Use – for a single line comment, and (* *) for multiline comment.
3. Facility declaration, file inclusions, main operation, procedure declaration, etc.
4. Operator ":=" is assignment, while "=" is check for equality.
5. Write ("RESOLVE is gr8"); or Write_Line ("RESOLVE is gr8");.
6. Write(some_string) prints the string and does not go to the next line, while Write_Line(some_string) prints the string and the newline character.
7. Translate it to Java, then compile and run Java.
8. Concepts, Facilities, Realizations, Enhancements.
9. Requires is a pre-condition to the operation, ensures is the post-condition.
10. a 2, b 4, c 1, d 3.
11. F, F, T, T.
12. My_Test_Facility.fa
13. Operation Mystery (evaluates Temp: Integer) : Boolean;
14. str is " ", and i is 6.
15. int1 is 17, and int2 is 11.

FUTURE NOTES:

Chapter 5 (from file Manual draft4)

Discuss convention and correspondence with example.
Discuss internal and external specs. Examples.

Chapter6

Complex Data Types
Stacks
Queues
Pr. Queue
Linked Lists
Basic_Map
Trees