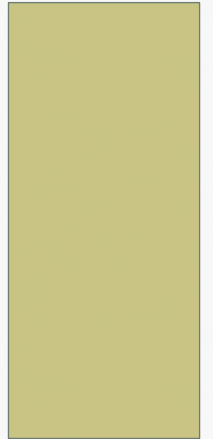


# WRITING RESOLVE CONCEPTS

STEPHEN SCHAUB

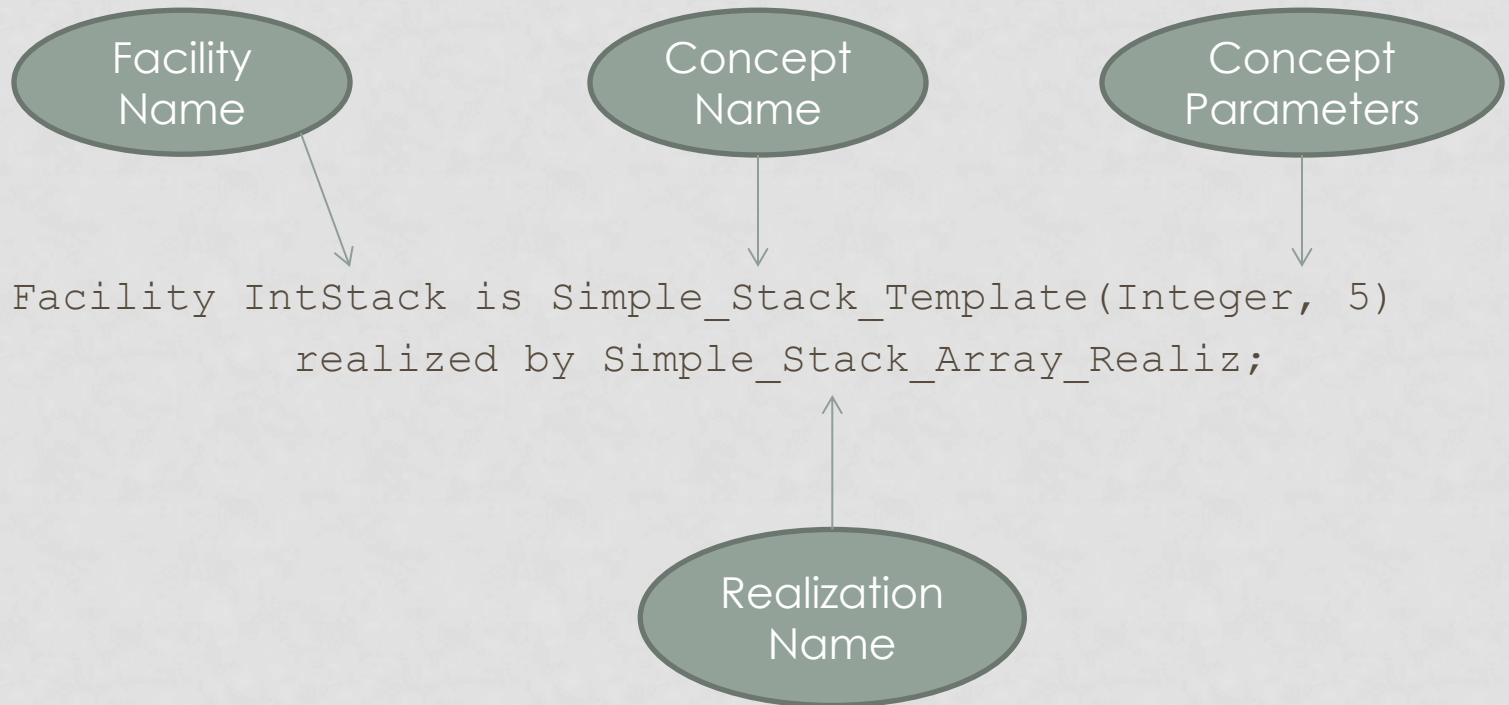


# RESOLVE TERMINOLOGY REVIEW

- A component in RESOLVE consists of
  - Concept – specifies component \_\_\_\_\_
  - Realization – provides component \_\_\_\_\_

# USING A RESOLVE COMPONENT

- Instantiate component with a Facility statement



# USING A RESOLVE COMPONENT

- Define a variable of the type specified by the Concept

```
Var S : IntStack.Stack;
```

- Call operations defined by the Concept

```
Num := 3;
```

```
IntStack.Push (Num, S) ;
```

- The facility name is usually optional

```
Var S : Stack;
```

```
Push (Num, S) ;
```

# UNDERSTANDING CONCEPT OPERATIONS

- How do you know what the operations do?
- Simple\_Stack\_Template has informal specifications
- Most RESOLVE concepts specify operation behavior using formal specifications

# MATHEMATICAL MODELING



# MODELING

- A model is a representation of something
- In Computer Science and Mathematics, consists of
  - Data abstraction
  - Operations that manipulate the data
- Example: Light switch
  - Two states
  - Data can be modeled in Java by ?
  - Operations: turn\_on, turn\_off, is\_on

# DATA TYPES

- Programming language data types are models of math sets

C++	Math
int	$\mathbb{Z}$
unsigned int	$\mathbb{N}$
double	$\mathbb{R}$
bool	$\mathbb{B}$
vector<E>	string of E
set<E>	set of E



# LAYERS OF MODELING

- A programming language data type (int) is a finite representation (model) of an infinite mathematical set ( $\mathbb{Z}$ )
- Mathematical types usually represent a set of numbers ...
  - ... but can also be used to model other things
- Example:
  - $\mathbb{N}$  usually represents the set of natural numbers ( $0..\infty$ )
  - Can also represent
    - Unicode / Ascii Characters
    - Enumerated values

# RESOLVE DATA TYPES

- Every RESOLVE data type is defined by a
  - Concept
  - Realization
- Concept specifies math model
  - Data types defined in terms of math types
  - Operations specified using math operations
- Realization provides implementation of model

# MATH TYPES IN RESOLVE

RESOLVE Math Type	Math Type
$\mathbb{Z}$	$\mathbb{Z}$
$\mathbb{N}$	$\mathbb{N}$
$\mathbb{B}$	$\mathbb{B}$
$\text{str}(E)$	string of $E$
$\text{powerset}(E)$	set of $E$

# DEFINING RESOLVE DATA TYPES

- A Concept uses the **Type Family** statement to define a new data type

```
Concept Integer_Template;  
  Type Family Integer is modeled by Z;  
  ...  
end;
```

- **Type Family** defines a data type (*Integer*) in terms of a mathematical type (*Z*)

# MODELING A LIGHT BULB

```
Concept LightBulb_Template;  
  Type Family LightBulb is modeled by B;  
    exemplar some_bulb;  
    initialization ensures some_bulb = false;  
    ...  
end;
```

- **initialization ensures** clause specifies initial value of this type
- **exemplar** defines a name that is referenced by **initialization ensures**

# ANOTHER LIGHT BULB MODEL

```
Concept LightBulb_Template;  
  Type Family LightBulb is modeled by N;  
    exemplar some_bulb;  
    constraint some_bulb <= 1;  
    initialization ensures _____;  
    ...  
end;
```

- **constraint** clause specifies restrictions on legal values
- operations can assume that the constraint is satisfied as a precondition, and must ensure the constraint is satisfied upon return

# TRAFFIC LIGHT

- How would you model a traffic light in a Java program?

# TRAFFIC LIGHT IN RESOLVE

- You could use  $N$  (constrained to 3 values)
- Or, use a cartesian product of three booleans:

Type Family `Traffic_Light` is modeled by `Cart_Prod`

```
    red: B;  
    yellow: B;  
    green: B;  
end;  
exemplar TL;  
constraint (TL.red and not TL.yellow and not TL.green)  
           or (not TL.red and TL.yellow and not TL.green)  
           or (not TL.red and not TL.yellow and TL.green);  
initialization ensures  
    TL.red and not TL.yellow and not TL.green;
```



# MATH MODELS FOR COLLECTIONS

- Need mathematical models for structures like lists, stacks, queues
- Useful math theory concepts:
  - Set
    - Example:  $\{ 5, 10, 15 \}$
    - Unordered collection
  - String
    - Example:  $\langle 5, 10, 15 \rangle$
    - Ordered collection
- Which kind is appropriate for lists / stacks / queues?

# MATH STRINGS

- Basic Math String operators in Resolve:
  - `| S |`
    - length of string `S`
  - `S1 ~ S2`
    - concatenate `S1` and `S2`
  - `<E>`
    - create a string containing `E`
  - `Reverse (S)`
    - Reverses `S`
- Math String constant:
  - `empty_string`
- Note: These can be used only in a **math theory context**

# A STACK MODEL

```
Concept Stack_Template(type Entry;  
                        evaluates Max_Depth: Integer);
```

```
Type Family Stack is modeled by Str(Entry);
```

```
  exemplar S;
```

```
  constraint _____;
```

```
  initialization ensures _____;
```

```
  ...
```

```
end;
```

# OPERATION SPECIFICATIONS



# REMEMBER 215?

```
public interface Stack<E> {  
    /*!  
     ** modeled_by: string of E  
     ** initial_value: <>  
    !*/  
  
    void push(E e);  
    /*!  
     ** preserves: e  
     ** ensures: self = <e> * #self  
    !*/  
  
    E pop();  
    /*!  
     ** requires: |self| > 0  
     ** ensures: #self = <pop()> * self  
    !*/  
  
    int length();  
    /*!  
     ** preserves: self  
     ** ensures: length() = |self|  
    !*/  
  
    void clear();  
    /*!  
     ** ensures: self = <>  
    !*/  
}
```

# FORMAL SPECIFICATION

From Stack\_Template:

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

- **requires** – specifies preconditions
  - |S| - length of initial S
  - Note: In **requires** clause, S really means #S
- **ensures** – specifies postconditions
  - #S refers to initial value of S
  - S refers to final value of S

# PARAMETER MODES REVISITED

- Recall that **preserves**, **restores**, and **evaluates** indicate that the parameter will not be modified upon return
- These modes automatically add  
*parameter = #parameter*  
to the ensures clause

# PARAMETER MODES REVISITED

- **alters**  $S$ : *type*
  - The operation modifies  $S$  in an unspecified way
  - $S$  is not mentioned in the ensures clause ( $\#S$  may be)
- **replaces**  $S$ : *type*
  - The operation modifies  $S$  to a value specified in the ensures clause
  - The incoming value is not used ( $S$  does not appear in requires clause)
- **updates**  $S$ : *type*
  - The operation modifies  $S$  to a value specified in the ensures clause that is based on  $\#S$
  - $S$  may appear in requires clause
- **clears**  $S$ : *type*
  - The operation must set  $S$  to the default value for its type



# FOR DISCUSSION

- Review Stack\_Template operations
- Why does Push use **alters**?
- Could Pop() return its value instead?

# FOR PRACTICE

- Define Traffic\_Light\_Concept using two booleans instead of three