

# Construction and Verification of Software

## 2019 - 2020

**MIEI - Integrated Master in Computer Science and Informatics**  
Consolidation block

**Lecture 3 - Specification and Verification (cont.)**

**João Costa Seco** ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))

based on previous editions by **Luís Caires** ([lcaires@fct.unl.pt](mailto:lcaires@fct.unl.pt))



FACULDADE DE  
CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE NOVA DE LISBOA

# Outline

---

- Hoare Logic revisited (Axiomatic approach)
- Algorithmic approach to verification
- Iteration (Loop Invariants) - 3 parts
- Abstract Data Types
- Objects as ADTs (in Dafny)

# Construction and Verification of Software

## 2019 - 2020

**MIEI - Integrated Master in Computer Science and Informatics**  
Consolidation block

### Lecture 3 - Part I - Hoare Logic Revisited

**João Costa Seco** ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))  
based on previous editions by **Luís Caires** ([lcaires@fct.unl.pt](mailto:lcaires@fct.unl.pt))



FACULDADE DE  
CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE NOVA DE LISBOA

# Simple Programming Language

$E ::=$	Expressions	
	$num$	Integer
	$x$	Variable
	$E + E \mid \dots$	Integer operators
	$E < E \mid \dots$	Relational operators
	$E \text{ and } E \dots$	Boolean operators

$P ::=$	Programs
$\text{skip}$	No op
	Assignment
$P; P$	Sequential Composition
	Conditional
$\text{while } E \text{ do } P$	Iteration

# Hoare Logic - Structural Rules

$$\{A\} \text{ skip } \{A\}$$

$$\{A[E/x]\} \ x := E \ {A}$$

$$\frac{\{A\} \ P \ \{B\} \quad \{B\} \ Q \ \{C\}}{\{A\} \ P; Q \ \{C\}}$$

$$\frac{A' \implies A \quad \{A\} \ P \ \{B\} \quad B \implies B'}{\{A'\} \ P \ \{B'\}}$$

# Rule for Assignment

---

$$\{A[E/x]\} \ x := E \ \{A\}$$

- $A[E/x]$  means:
  - the result of replacing all free occurrences of variable  $x$  in assertion  $A$  by the expression  $E$
  - For this rule to be sound, we require  $E$  to be an expression without side effects (a pure expression)

# Rule for Assignment

---

$$\{A[E/x]\} \ x := E \ \{A\}$$

- We can think of A as a condition where “x” appears in some places. A is a condition dependent on “x”.
- The assignment  $x := E$  changes the value of x to E, but leaves everything else unchanged
- So everything that could be said of E in the precondition, can be said of x in the postcondition, since the value of x after the assignment is E
- Example:  $\{x + 1 > 0\} x := x + 1 \ {x > 0\}$

# Rule for Assignment

---

$$\{A[E/x]\} \ x := E \ {A}$$

- Example, let's check  $\{x > -1\} x := x + 1 \{x > 0\}$

$\{ (x+1 > 0) \} x := x+1 \ {x > 0} \quad \text{by the } := \text{ Rule}$

that is,  $\{ (x > 0)[x+1/x] \} \ x := (x+1) \ {x > 0}$

$\{x > -1\} x := x + 1 \ {x > 0} \quad \text{by deduction}$

# Rule for Assignment

---

$$\{A[E/x]\} \ x := E \ \{A\}$$

- Trick: if  $x$  does not appear in  $E$  or  $A$ .

We can always write  $\{ A \&& E == E \} x := E \{ x == E \}$

So, if  $x$  does not occur in  $E$ ,  $A$  the triple

$$\{ A \} x := E \{ A \&& x == E \}$$

is always valid

# Rule for Assignment

---

$$\{A[E/x]\} \ x := E \ \{A\}$$

- Exercises. Derive:
  - $\{y > 0\} x := y \{x > 0 \&\& y == x\}$
  - $\{x == y\} x := 2^*x \{y == x \text{ div } 2\}$
  - $\{P(y) \&\& Q(z)\}$  (here P and Q are any properties)  
 $x := y ; y := z; z := x$   
 $\{P(z) \&\& Q(y)\}$

# Exercises

---

Prove using the assignment rule that:

```
assert y > 0;  
x := y;  
assert x > 0 && y == x;
```

```
assert y == x;  
x := 2 * x;  
assert y == x / 2;
```

# Exercises

---

Prove using the assignment rule that:

```
function P(x:int):bool {  
    ...  
}  
function Q(x:int):bool {  
    ...  
}  
  
var x := ...;  
var y := ...;  
var z;  
  
assert P(x) && Q(y);  
z := x;  
x := y;  
y := z;  
assert P(y) && Q(x);
```

# Example

---

- Consider the program

$$P \triangleq \text{if } (x > y) \text{ then } z := x \text{ else } z := y$$

- We (mechanically) check the triple

$$\{ \text{true} \} P \{ z == \max(x,y) \}$$

# Example

---

- Consider the program

$$P \triangleq \text{if } (x > y) \text{ then } z := x \text{ else } z := y$$

- We (mechanically) check the triple

$$\{ \text{true} \} P \{ z == \max(x,y) \}$$
$$\{ x == \max(x,y) \} \ z := x \{ z == \max(x,y) \}$$
$$\{ x > y \} \ z := x \{ z == \max(x,y) \}$$
$$\{ y == \max(x,y) \} z := y \{ z == \max(x,y) \}$$
$$\{ y >= x \} z := y \{ z == \max(x,y) \}$$

# Construction and Verification of Software

## 2019 - 2020

**MIEI - Integrated Master in Computer Science and Informatics**  
Consolidation block

### **Lecture 3 - Part II - Weakest Precondition Algorithm**

**João Costa Seco** ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))  
based on previous editions by **Luís Caires** ([lcaires@fct.unl.pt](mailto:lcaires@fct.unl.pt))



**FACULDADE DE  
CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE NOVA DE LISBOA**

# The Weakest Precondition

Programming  
Languages

T.A. Standish  
Editor

## Guarded Commands, Nondeterminacy and Formal Derivation of Programs

Edsger W. Dijkstra  
Burroughs Corporation

So-called “guarded commands” are introduced as a building block for alternative and repetitive constructs that allow nondeterministic program components for which at least the activity evoked, but possibly even the final state, is not necessarily uniquely determined by the initial state. For the formal derivation of programs expressed in terms of these constructs, a calculus will be shown.

**Key Words and Phrases:** programming languages, sequencing primitives, program semantics, programming language semantics, nondeterminacy, case-construction, repetition, termination, correctness proof, derivation of programs, programming methodology

**CR Categories:** 4.20, 4.22

### 1. Introduction

In Section 2, two statements, an alternative construct and a repetitive construct, are introduced, together with an intuitive (mechanistic) definition of their semantics. The basic building block for both of them is the so-called “guarded command,” a statement list prefixed by a boolean expression: only when this boolean expression is initially true, is the statement list eligible for execution. The potential nondeterminacy allows us to map otherwise (trivially) different programs on the same program text, a circumstance that seems largely responsible for the fact that programs can now be derived in a manner more systematic than before.

In Section 3, after a prelude defining the notation, a formal definition of the semantics of the two constructs is given, together with two theorems for each of the constructs (without proof).

In Section 4, it is shown how, based upon the above, a formal calculus for the derivation of programs can be founded. We would like to stress that we do not present “an algorithm” for the derivation of programs: we have used the term “a calculus” for a formal discipline—a set of rules—such that, if applied successfully: (1) it will have derived a correct program; and (2) it will tell us that we have reached such a goal. (We use the term as in “integral calculus.”)

# Predicate transformer semantics

---

- Algorithmic approach to verify a while program
- Defines a predicate transformer that produces the weakest precondition for a pair of program/assertion

$$wp(P, B)$$

- Any Hoare triple  $\{A\} P \{B\}$  is provable if and only if the predicate  $A \Rightarrow wp(P, B)$  holds.
- The predicate is recursively defined on the cases of the program syntax.

# Predicate transformer semantics

---

$$\{A\} \text{ skip } \{A\}$$

$$wp(\text{skip}, A) = A$$

$$\frac{\{A\} P \{B\} \quad \{B\} Q \{C\}}{\{A\} P; Q \{C\}}$$

$$wp(P; Q, C) = wp(P, wp(Q, C))$$

$$\{A[E/x]\} x := E \{A\}$$

$$wp(x := E, A) \triangleq A[E/x]$$

$$\frac{\{A \wedge E\} P \{B\} \quad \{A \wedge \neg E\} Q \{B\}}{\{A\} \text{ if } E \text{ then } P \text{ else } Q \{B\}}$$

$$wp(\text{if } E \text{ then } P_1 \text{ else } P_2, B) \triangleq \\ E \Rightarrow wp(P_1, B) \wedge \neg E \Rightarrow wp(P_2, B)$$

# Example (again)

---

- Consider the program

$$P \triangleq \text{if } (x > y) \text{ then } z := x \text{ else } z := y$$

- We (mechanically) check the triple

$$\{ \text{true} \} P \{ z == \max(x,y) \}$$

# Example (again)

---

- Consider the program

$P \triangleq x := y ; y := z; z := x$

We (mechanically) check the triple

{  $P(y) \&& Q(z)$  }  $P$  {  $P(z) \&& Q(y)$  }

(here  $P$  and  $Q$  are any properties)

# Construction and Verification of Software

## 2019 - 2020

**MIEI - Integrated Master in Computer Science and Informatics**  
Consolidation block

### Lecture 3 - Part III - Loop invariants

**João Costa Seco** ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))  
based on previous editions by **Luís Caires** ([lcaires@fct.unl.pt](mailto:lcaires@fct.unl.pt))



FACULDADE DE  
CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE NOVA DE LISBOA

# Rule for Iteration

---

$$\frac{\{? \wedge E\} \ P \ \{?\}}{\{A\} \text{ while } E \text{ do } P \ \{\neg E \wedge ?\}}$$

Any precise post condition depends on how many times P is executed ... P can be executed 0, 1, 2 ... n times, n not really known at verification time.

# Rule for Iteration

---

```
while  $E$  do  $P \triangleq$ 
    if  $E$  then  $P;$ 
        if  $E$  then  $P;$ 
            if  $E$  then  $P, \dots$ 
                else skip
            else skip
        else skip
    else skip
```

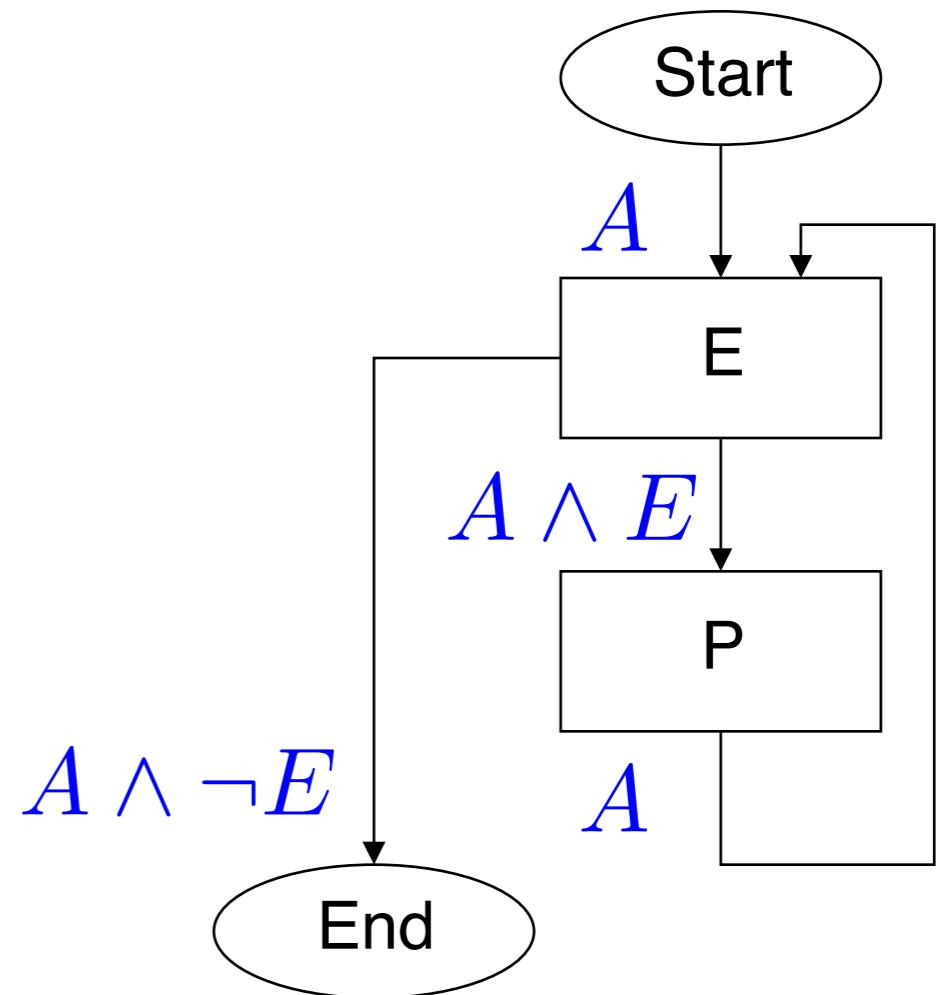
# Rule for Iteration

---

$$\{A\} \text{while } E \text{ do } P \{B\} \triangleq$$
$$\{A\} \text{if } E \text{ then } \{A \wedge E\} P \{B_1\};$$
$$~~~~~\{B_1\} \text{if } E \text{ then } \{B_1 \wedge E\} P \{B_2\};$$
$$~~~~~\{B_2\} \text{if } E \text{ then } \{B_2 \wedge E\} P \{B_3\};$$
$$~~~~~\dots$$
$$~~~~~\text{else skip } \{\neg E \wedge B_2\}$$
$$~~~~~\text{else skip } \{\neg E \wedge B_1\}$$
$$~~~~~\text{else skip } \{\neg E \wedge A\}$$

# Rule for Iteration

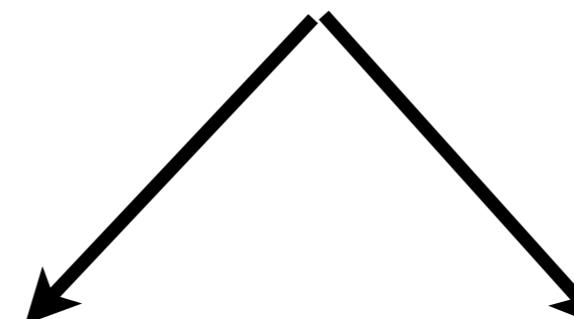
$$\frac{\{A \wedge E\} \ P \ \{A\}}{\{A\} \text{ while } E \text{ do } P \ \{A \wedge \neg E\}}$$



# Rule for Iteration

---

INV = Invariant Condition

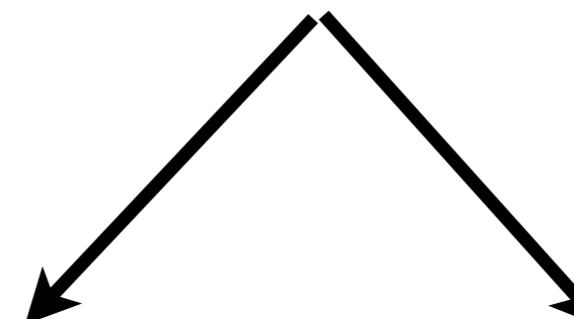
$$\frac{\{ \text{Inv} \wedge E \} \ P \ \{ \text{Inv} \}}{\{ \text{Inv} \} \text{ while } E \text{ do } P \ \{ \text{Inv} \wedge \neg E \}}$$


- We cannot predict in general how many iterations will the while loop do (undecidability of the halting problem).
- We approximate all iterations by an **invariant condition**
- A loop invariant is a condition that holds at loop entry and at loop exit.

# Rule for Iteration

---

INV = Invariant Condition

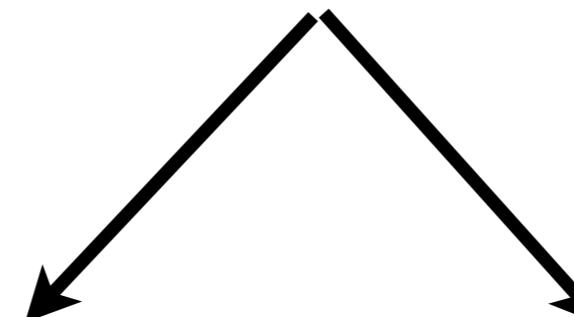
$$\frac{\{ \text{Inv} \wedge E \} \ P \ \{ \text{Inv} \}}{\{ \text{Inv} \} \text{ while } E \text{ do } P \ \{ \text{Inv} \wedge \neg E \}}$$


- If the invariant holds initially and is preserved by the loop body, it will hold when the loop terminates!
- It does not matter how many iterations will run
- Unlike for other rules of Hoare logic, finding the invariant requires human intelligence (you are a programmer :-)

# Rule for Iteration

---

INV = Invariant Condition

$$\frac{\{ \text{Inv} \wedge E \} \ P \ \{ \text{Inv} \}}{\{ \text{Inv} \} \text{ while } E \text{ do } P \ \{ \text{Inv} \wedge \neg E \}}$$


- The invariant depicts the state in all iterations of a loop.
- The invariant works like the induction hypothesis in a proof. The base case is the loop executed 0 times, the loop body is the induction step that iterates from step n to n+1. There must exist a valid induction measure.

# Algorithmic approach for Iteration

---

$$\begin{aligned} wp(\text{while } E \text{ do } P, B) &\triangleq \\ I \wedge (E \wedge I \Rightarrow wp(P, I)) \wedge (\neg E \wedge I \Rightarrow B) \end{aligned}$$

# Loop Invariants

---

```
{0 ≤ n}  
i := 0;  
while i < n do {  
    i := i + 1  
}  
{i == n}
```

# Loop Invariants

---

```
{0 ≤ n}  
i := 0;  
{i == 0 ∧ 0 ≤ n}  
{0 ≤ i ≤ n}  
while i < n do {  
    {0 ≤ i ≤ n ∧ i < n}  
    {0 ≤ i < n}  
    {0 ≤ i + 1 ≤ n}  
    i := i + 1  
    {0 ≤ i ≤ n}  
}  
{0 ≤ i ≤ n ∧ i >= n}  
{i == n}
```

# Construction and Verification of Software

## 2019 - 2020

**MIEI - Integrated Master in Computer Science and Informatics**  
Consolidation block

**Lecture 3 - Part IV - Breaking and Fixing Loop invariants**

**João Costa Seco** ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))  
based on previous editions by **Luís Caires** ([lcaires@fct.unl.pt](mailto:lcaires@fct.unl.pt))



FACULDADE DE  
CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE NOVA DE LISBOA

# Loop Invariants

---

- Consider program P defined by

$$P \triangleq s := 0; i := 0; \text{while } i < n \text{ do } \{i := i + 1; s := s + i\}$$

- What is the specification of P? What does P do?

$$\{A\} P \{B\}$$

# Loop Invariants

---

- Consider program P defined by

$$P \triangleq s := 0; i := 0; \text{while } i < n \text{ do } \{i := i + 1; s := s + i\}$$

- What is the specification of P? What does P do?

$$\{n \geq 0\} \ P \ \{s = \sum_{j=0}^n j\}$$

Is this a good specification for program P?

Can we mechanically check the Hoare triple?

# Loop Invariants

---

$$\{0 \leq n\}$$

$s := 0;$

$i := 0;$

**while**  $i < n$  **do** {

$i := i + 1;$

$s := s + i$

}

$$\{s == \sum_j^n j\}$$

# Loop Invariants

---

$\{0 \leq n\}$

$s := 0;$

$\{s = 0 \wedge 0 \leq n\}$

$i := 0;$

$\{s = 0 \wedge 0 \leq i \leq n\}$

**while**  $i < n$  **do** {

$i := i + 1;$

$s := s + i$

}

$\{s = \sum_j^n j\}$

# Loop Invariants

---

$\{0 \leq n\}$

$s := 0;$

$\{s = 0 \wedge 0 \leq n\}$

$i := 0;$

$\{s = 0 \wedge 0 \leq i \leq n\}$

$\{0 \leq i \leq n \wedge s = \sum_{j=0}^i \cdot j\}$

**while**  $i < n$  do {

$i := i + 1;$

$s := s + i$

}

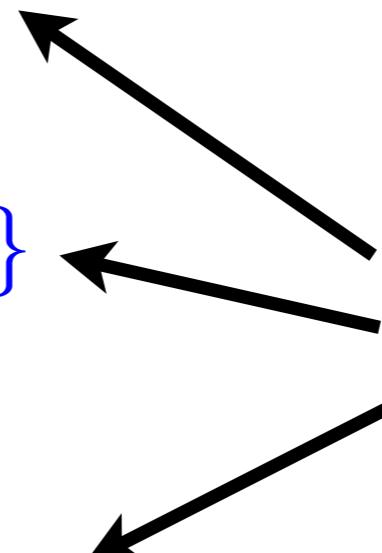
$\{i = n \wedge s = \sum_{j=0}^i \cdot j\}$

$\{s = \sum_{j=0}^n \cdot j\}$

# Loop Invariants

---

```
{0 ≤ n}  
s := 0;  
{s = 0 ∧ 0 ≤ n}  
i := 0;  
{s = 0 ∧ i = 0 ∧ 0 ≤ i ≤ n}  
{0 ≤ i ≤ n ∧ s =  $\sum_{j=0}^i j$ }  
while i < n do {  
    {0 ≤ i ≤ n ∧ s =  $\sum_{j=0}^i j$ }  
    i := i + 1;  
    s := s + i  
    {0 ≤ i ≤ n ∧ s =  $\sum_{j=0}^i j$ }  
}  
{i = n ∧ s =  $\sum_{j=0}^n j$ }  
{s =  $\sum_{j=0}^n j$ }
```



Invariant holds

# Loop Invariants

---

- The loop invariant may be broken inside the body of the loop, but must be re-established at the end.
- Notice the assignment rule

$$\{A[E/x]\} \quad x := E \quad \{A\}$$

- that breaks the invariant...

$$\{0 \leq i \leq n \wedge i < n \wedge s = \sum_{j=0}^i \cdot j\}$$

$$\{0 \leq i < n \wedge s = \sum_{j=0}^i \cdot j\}$$

$$i := i + 1$$

$$\{0 \leq i - 1 < n \wedge s = \sum_{j=0}^{i-1} \cdot j\}$$

$$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i-1} \cdot j\}$$

# Loop Invariants

---

- The loop invariant may be broken inside the body of the loop, but must be re-established at the end.
- Notice the assignment rule

$$\{A[E/x]\} \quad x := E \quad \{A\}$$

- and then re-establishes it

$$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i-1} j\}$$

$$s := s + i$$

$$\{0 \leq i \leq n \wedge s = (\sum_{j=0}^{i-1} j) + i\}$$

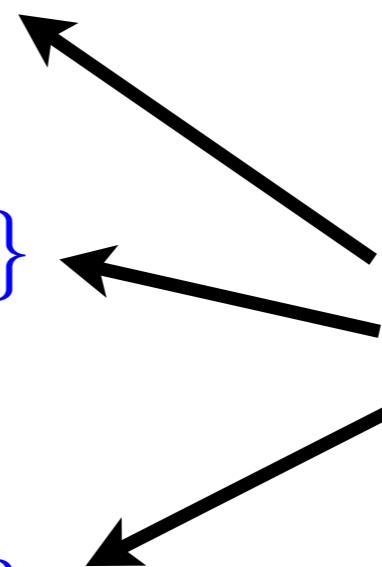
$$\{0 \leq i \leq n \wedge s = (\sum_{j=0}^i j)\}$$

# Loop Invariants

---

```
{0 ≤ n}  
s := 0;  
{s = 0 ∧ 0 ≤ n}  
i := 0;  
{s = 0 ∧ i = 0 ∧ 0 ≤ i ≤ n}  
{0 ≤ i ≤ n ∧ s =  $\sum_{j=0}^i j$ }  
while i < n do {  
    {0 ≤ i ≤ n ∧ s =  $\sum_{j=0}^i j$ }  
    i := i + 1;  
    s := s + i  
    {0 ≤ i ≤ n ∧ s =  $\sum_{j=0}^i j$ }  
}  
{i = n ∧ s =  $\sum_{j=0}^n j$ }  
{s =  $\sum_{j=0}^n j$ }
```

Invariant holds



# Loop Invariants

$$\{0 \leq n\}$$

$s := 0;$

$$\{s = 0 \wedge 0 \leq n\}$$

$i := 0;$

$$\{s = 0 \wedge i = 0 \wedge 0 \leq i \leq n\}$$

$$\{0 \leq i \leq n \wedge s = \sum_{j=0}^i j\}$$

**while**  $i < n$  **do** {

$$\{0 \leq i \leq n \wedge i < n \wedge s = \sum_{j=0}^i j\}$$

$$\{0 \leq i < n \wedge s = \sum_{j=0}^i j\}$$

$i := i + 1;$

$$\{0 \leq i - 1 < n \wedge s = \sum_{j=0}^{i-1} j\}$$

$$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i-1} j\}$$

$s := s + i$

$$\{0 \leq i \leq n \wedge s = (\sum_{j=0}^{i-1} j) + i\}$$

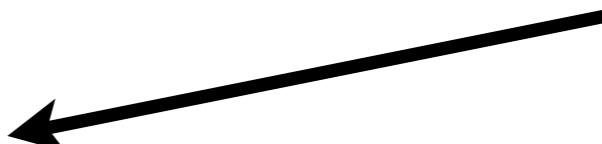
$$\{0 \leq i \leq n \wedge s = \sum_{j=0}^i j\}$$

}

$$\{i = n \wedge s = \sum_{j=0}^i j\}$$

$$\{s = \sum_j^n j\}$$

Invariant  
broken



Invariant  
restored



# Loop Invariants

---

$$\{0 \leq n\}$$

$s := 0;$

$$\{s = 0 \wedge 0 \leq n\}$$

$i := 0;$

$$\{s = 0 \wedge i = 0 \wedge 0 \leq i \leq n\}$$

$$\{0 \leq i \leq n \wedge s = \sum_{j=0}^i j\}$$

**while**  $i < n$  **do** {

$$\{0 \leq i \leq n \wedge i < n \wedge s = \sum_{j=0}^i j\}$$

$$\{0 \leq i < n \wedge s = \sum_{j=0}^i j\}$$

$i := i + 1;$

$$\{0 \leq i - 1 < n \wedge s = \sum_{j=0}^{i-1} j\}$$

$$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i-1} j\}$$

$s := s + i$

$$\{0 \leq i \leq n \wedge s = (\sum_{j=0}^{i-1} j) + i\}$$

$$\{0 \leq i \leq n \wedge s = \sum_{j=0}^i j\}$$

}

$$\{i = n \wedge s = \sum_{j=0}^i j\}$$

$$\{s = \sum_j^n j\}$$

Invariant holds

The diagram consists of two black arrows. One arrow points from the equation  $\{0 \leq i \leq n \wedge s = \sum_{j=0}^i j\}$  to the final equation  $\{s = \sum_j^n j\}$ . Another arrow points from the equation  $\{0 \leq i \leq n \wedge s = \sum_{j=0}^i j\}$  to the equation  $\{s = \sum_{j=0}^n j\}$ .

# Hints for finding loop invariants

---

- **First:** carefully think about the post condition of the loop
  - Typically the post-condition talks about a property “accumulated” across a “range”  
(this is why you are using a loop, right?)
  - e.g., maximum of all elements of an array
  - e.g., sort visited elements in a data structure

# Hints for finding loop invariants

---

- **Second:** design a “generalised” version of the post-condition, in which the already visited part of the data is made explicit as a function of the “loop control variable”
- The loop body may temporarily break the invariant, but must restore it at the end of the body
- **Important:** make sure that the invariant together (`&&`) with the termination condition really implies your post-condition

# Examples, what kind of invariant we need for...

---

- **Max of an array**
  - All elements to the left are smaller than the max so far
- **Array Searching (unsorted)**
  - All elements left of the index are different from the value being searched
- **Array Searching (sorted)**
  - The element is between the lower and the higher limits
- **Sorting (bubblesort, insertion sort, etc.)**
  - Everything to the left of the cursor is sorted
- **List Reversing**
  - All elements to the left of the cursor are placed on the right of the result

# Construction and Verification of Software

## 2019 - 2020

**MIEI - Integrated Master in Computer Science and Informatics**  
Consolidation block

### Lecture 3 - Part V - Examples

**João Costa Seco** ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))  
based on previous editions by **Luís Caires** ([lcaires@fct.unl.pt](mailto:lcaires@fct.unl.pt))



FACULDADE DE  
CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE NOVA DE LISBOA

# Exercise - Fibonacci

---

```
function fib(n : int) : int
// this is the recursive spec of fibonacci
requires n>=0;
{
    if (n==0) then 1 else
        if (n==1) then 1 else fib(n-1)+fib(n-2)
}

// the method fibo below should implement fib efficiently
// “bottom up” using a while loop
method fibo(n : int) returns (f : int)
requires n>=0;
ensures f == fib(n);
{
    ...
}
```

# Exercise - Fibonacci

---

```
method Fib(n:int) returns (f:int)
    requires n >= 0
    ensures f == fib(n)
{
    if( n == 0 || n == 1 ) { return 1; }
    var a := 1;
    var b := 1;
    var i := 1;
    while i < n
        decreases n - i
        invariant 1 <= i <= n
        invariant a == fib(i-1)
        invariant b == fib(i)
    {
        a, b := b, a+b; // fib(i-1) + fib(i) = fib(i+1)
        i := i + 1;
    }
    f := b;
}
```

# Exercise

---

```
// return the maximum of the values in array a[-]  
// in positions i such that 0 <= i < n
```

```
method Max(a:array<int>, n:int) returns (m:int)
```

```
// write the code and fully check it with dafny  
// define the weakest preconditions you can think of  
// define the strongest postconditions you can think of
```

# Exercise - Max of an array

---

```
method maxArray(a:array<int>, n:int) returns (max:int)
    requires 0 < n <= a.Length
    ensures forall j :: 0 <= j < n ==> a[j] <= max
{
    max := a[0];
    var i := 1;
    while i < n
        ...
    {
        if max < a[i] {
            max := a[i];
        }
        i := i + 1;
    }
}
```

# Exercise 6 - daffy solution

---

```
method maxArray(a:array<int>, n:int) returns (max:int)
    requires 0 < n <= a.Length
    ensures forall j :: 0 <= j < n ==> a[j] <= max
{
    max := a[0];
    var i := 1;
    while i < n
        decreases n-i
        invariant 0 <= i <= n
        invariant forall j :: 0 <= j < i ==> a[j] <= max
    {
        if max < a[i] {
            max := a[i];
        }
        i := i + 1;
    }
}
```