

# *Real-Time & Embedded Systems 2019*



## Introduction & Languages

Uwe R. Zimmer - The Australian National University



# Introduction & Languages

## References

[Berry2000]

Berry, Gerard

*The Esterel v5 Language Primer -Version v5.91*

2000 pp. 1-148

Burns and Wellings. Concurrent and Real-Time Programming in Ada, edition. Cambridge University Press (2007)

[Burns2007]

Burns, Alan & Wellings, Andy  
*Concurrent and Real-Time Programming in Ada*  
Cambridge University Press (2007)

[Davies1995]

Davies, J & Schneider, S

*A brief history of Timed CSP*

Theoretical Computer Science 1995 vol. 138 (2) pp. 243-271

[Lee2009]

Lee, Edward

*Computing Needs Time*  
Technical Report No. UCB/EECS-2009-30 2009

[NN1995]

*Occam 2.1 reference manual*  
1995 pp. 1-171

[NN2004]

*The Real-Time Java Platform*  
A Technical White Paper  
June 2004 2004 pp. 1-26

[NN2006]

*Ada Reference Manual - Language and Standard Libraries*  
Ada Reference Manual  
ISO/IEC 8652:1995(E)

[Society2009]

Society, Design Automation Standards Committee of the IEEE Computer  
*IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*  
*IEEE Standard VHDL Language Reference Manual*  
2009 pp. 1-640

[Walli1995]

Walli, Stephen

*The POSIX family of standards*

*StandardView* 1995 vol. 3 (1)



# *Introduction & Languages*

*What is a real-time system?*

*Features of a Real-Time System?*

- Fast context switches?
- Small size?
- Quick responds to external interrupts?
- Multitasking?
- ‘Low level’ programming interfaces?
- Inter-process communication tools?
- High processor utilization?
- Fast systems?



# *Introduction & Languages*

## *What is a real-time system?*

## *Features of a Real-Time System?*

- ~~Fast context switches?~~ ➡ Should be fast anyway!
- ~~Small size?~~ ➡ Should be small anyway!
- ~~Quick responds to external interrupts?~~ ➡ Predictable! – not ‘quick’
- ~~Multitasking?~~ ➡ Real time systems are often multitasking systems
- ~~‘Low level’ programming interfaces?~~ ➡ Needed in many systems!
- ~~Inter-process communication tools?~~ ➡ Needed for any concurrency!
- ~~High processor utilization?~~ ➡ Just the opposite usually! (redundancy)
- ~~Fast systems?~~ ➡ Predictable! – not ‘fast’



# *Introduction & Languages*

## *What is a real-time system?*

### *Definition of a Real-Time System*

The correctness of a real-time systems depends on:

1. The **logical correctness** (accuracy) of the **results**

... as well as ...

2. The **time when** the result is **delivered**.

... both with respect to the specification.

Real-Time systems are frequently evaluated as part of a physical system.



# *Introduction & Languages*

## *What is a real-time system?*

### *Real-Time Systems Scenarios*

- **All sizes and complexities:** From heating regulators over mobile phones to high speed trains, aircraft, satellites, space station(s) ...
  - **Situated:** Almost always part of or coupled to a physical system.
  - **Relevant:** Vital components of our traffic and communication infrastructure among many other essential systems.
  - **Dangerous:** Failures often lead to loss of life, or environmental damage.
- ☞ Real-Time Systems require a specific understanding and skill set.



# *Introduction & Languages*

## *What is a real-time system?*

### *Typical characteristics of a Real-Time System*

- **Adherence to set time constraints**  
Not too early – not too late
- **Predictability**  
Repeatable results in time and value
- **Fault tolerance**  
Robustness in the presence of foreseeable faults
- **Accuracy**  
Results are precise enough to drive e.g. a physical systems
- **Frequently: concurrent, distributed  
and employing complex hardware.**



# *Introduction & Languages*

## *Simple example: Brake manager*

Latencies:

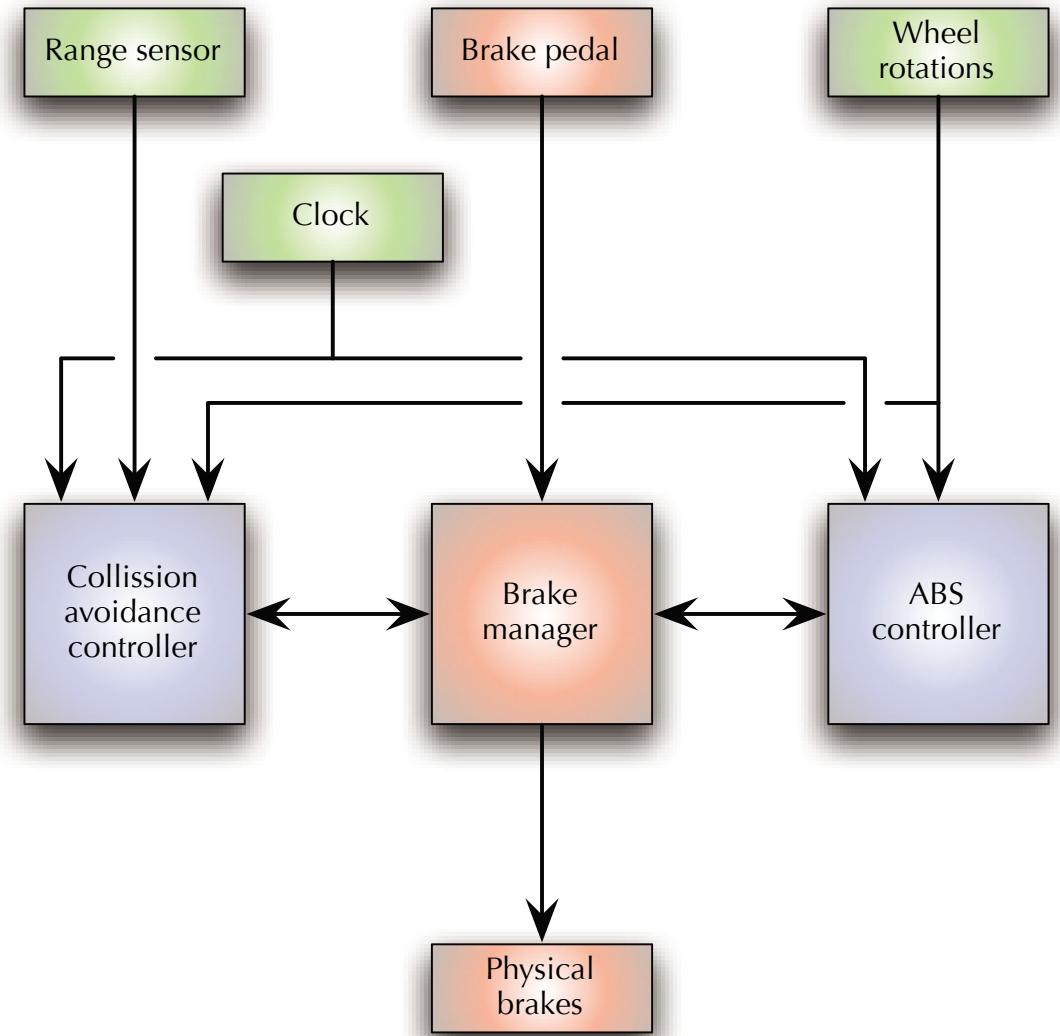
- Constant.
- Significantly shorter than the driver's response time.

Reliability:

- Provide robustness under all foreseeable and “manageable” failures.

Efficient design:

- Mass producible?





# *Introduction & Languages*

## *Simple example: Brake manager*

### *Ways to define reliability*

- Full fault tolerant, graceful degradation or fail safe?
- Redundancies?
- Testing?
- Verification?
- (Physical) Modularization?

☞ Use an algebraic / real-time logic tool?

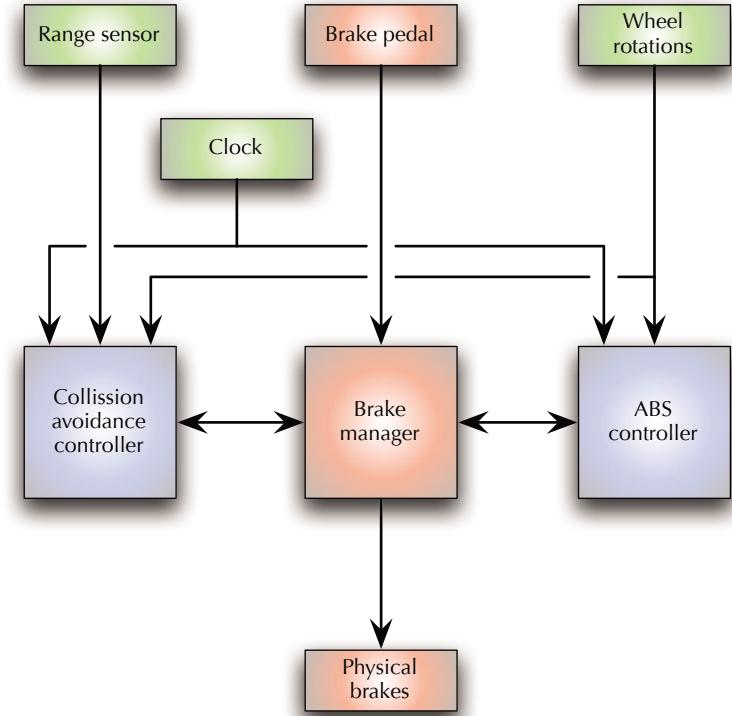
Proof correctness?

☞ Use a predictable runtime environment and language?

Certification? Test all relevant cases?

☞ Assume things will still go bad?

Provide fall-backs?





# *Introduction & Languages*

## *Simple example: Brake manager*

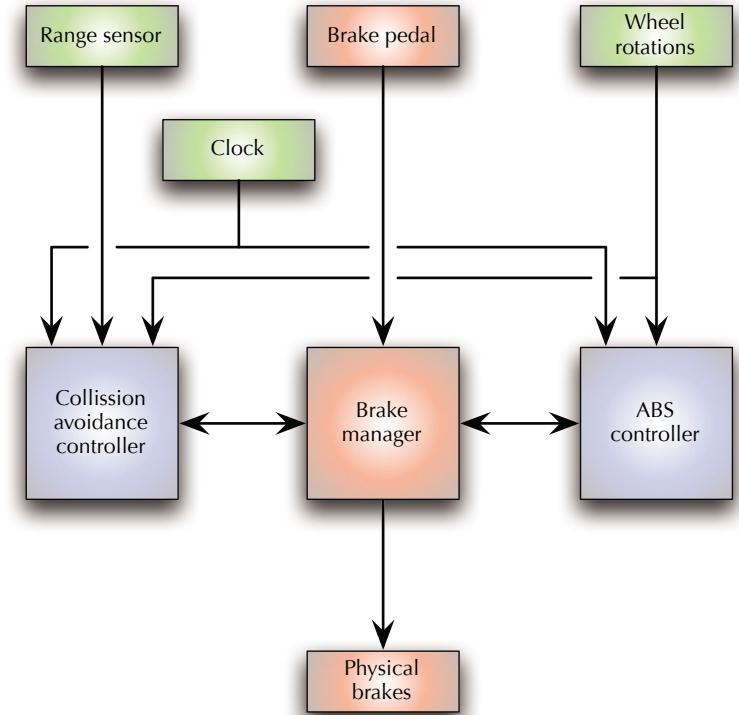
### *Ways to implement this (hardware)*

The brakes:

- Mechanical (hydraulic) + overwrite valves
- Digitally controlled, (no mechanical connection)
- Brake lights

The controller(s):

- Single CPU
- Multiple CPUs + shared memory
- Multiple CPUs + point-to-point connections
- Multiple CPUs + communication system (e.g. a bus system)
- Redundant CPUs



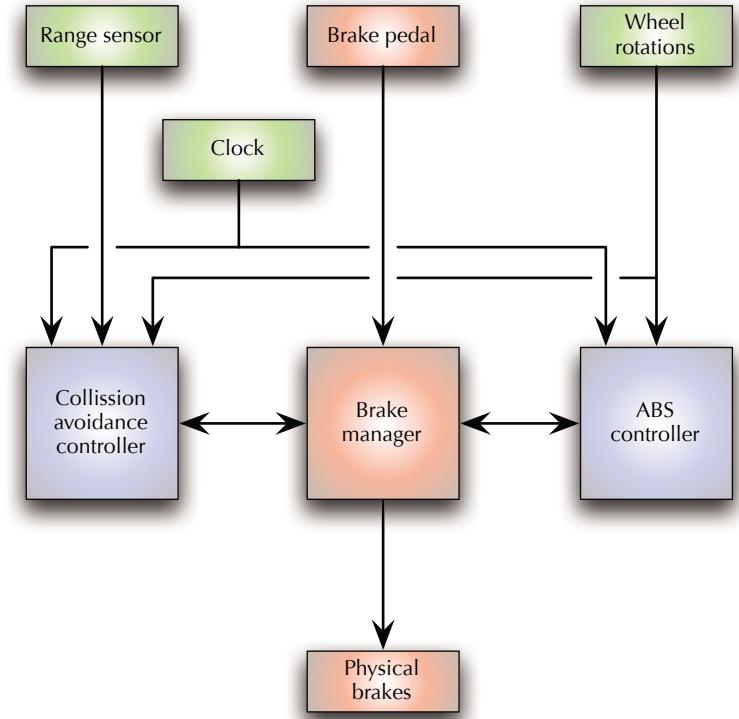


# *Introduction & Languages*

## *Simple example: Brake manager*

### *Ways to implement this (software)*

- Sequential, concurrent or distributed?
- Shared memory or message passing?
- Synchronous or asynchronous communication?
- Dynamic or fixed schedule?
- Imperative, functional, or dataflow programming?
- Predefined communication channels or client/server models?
- Data driven or (global) clock synchronized?
- Polling, interrupt driven, or event driven?
- Globally synchronous, individually synchronous, or asynchronous I/O channels?
- Languages/tools which lend themselves to verification and validation?
- Languages/tools which lend themselves to certification and accreditation?



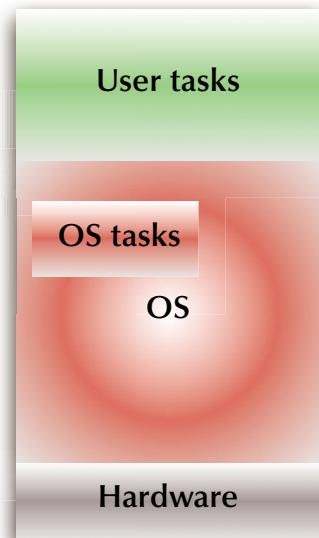


# *Introduction & Languages*

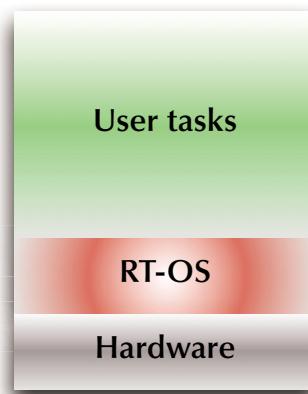
## *What is a real-time system?*

### *Typical Real-Time Operating System*

Often implemented as an integrated run-time environment, i.e. there is 'no operating system' (☞ embedded systems).



Typical  
"Standard" OS



Typical  
Real Time-OS



Typical  
Embedded system

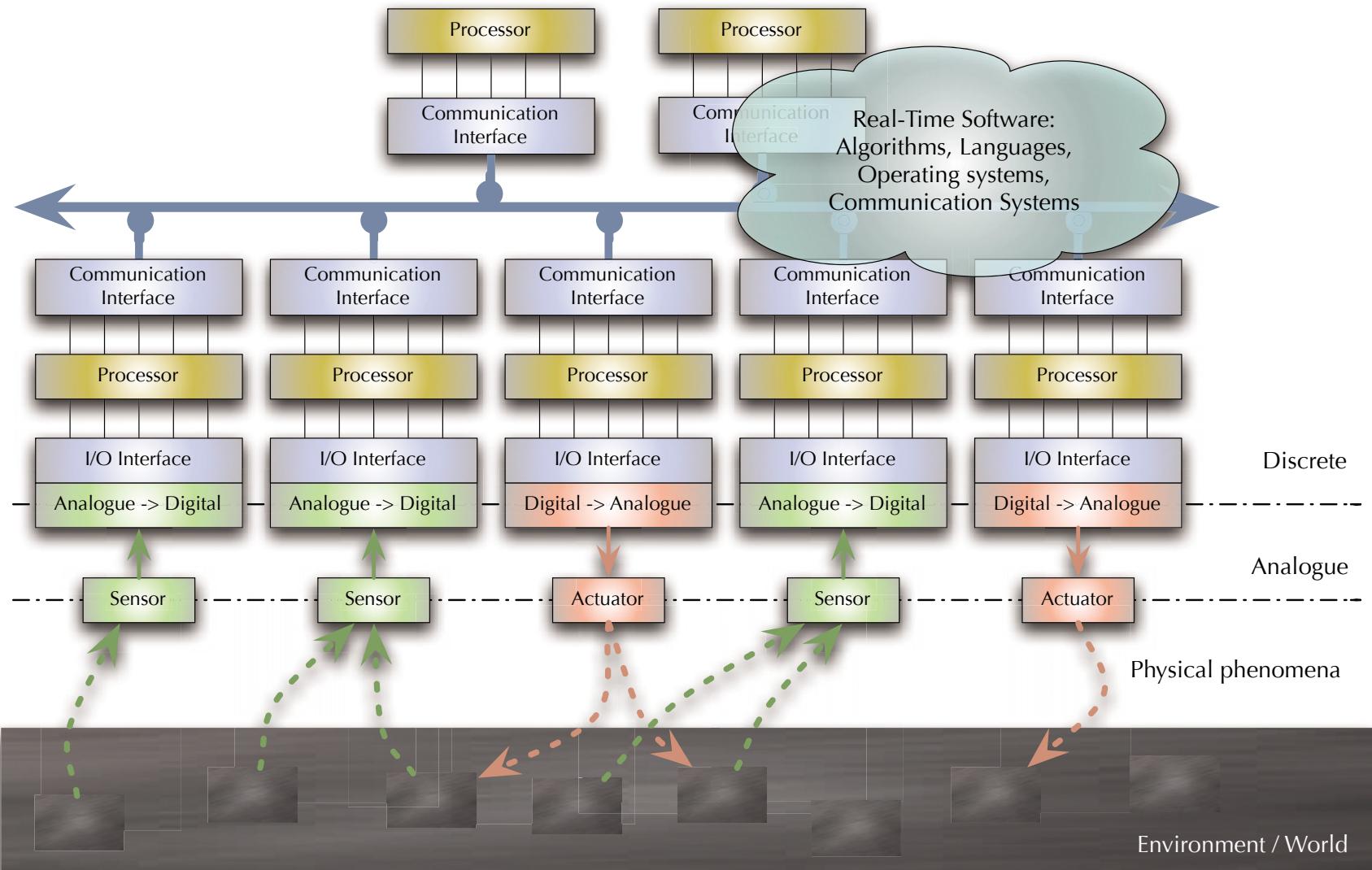
RT-OSS provide:

- ☞ **Predictability**
- ☞ **Passivity**
- ☞ **Small footprint**
- ☞ **Instrumentation**



# Introduction & Languages

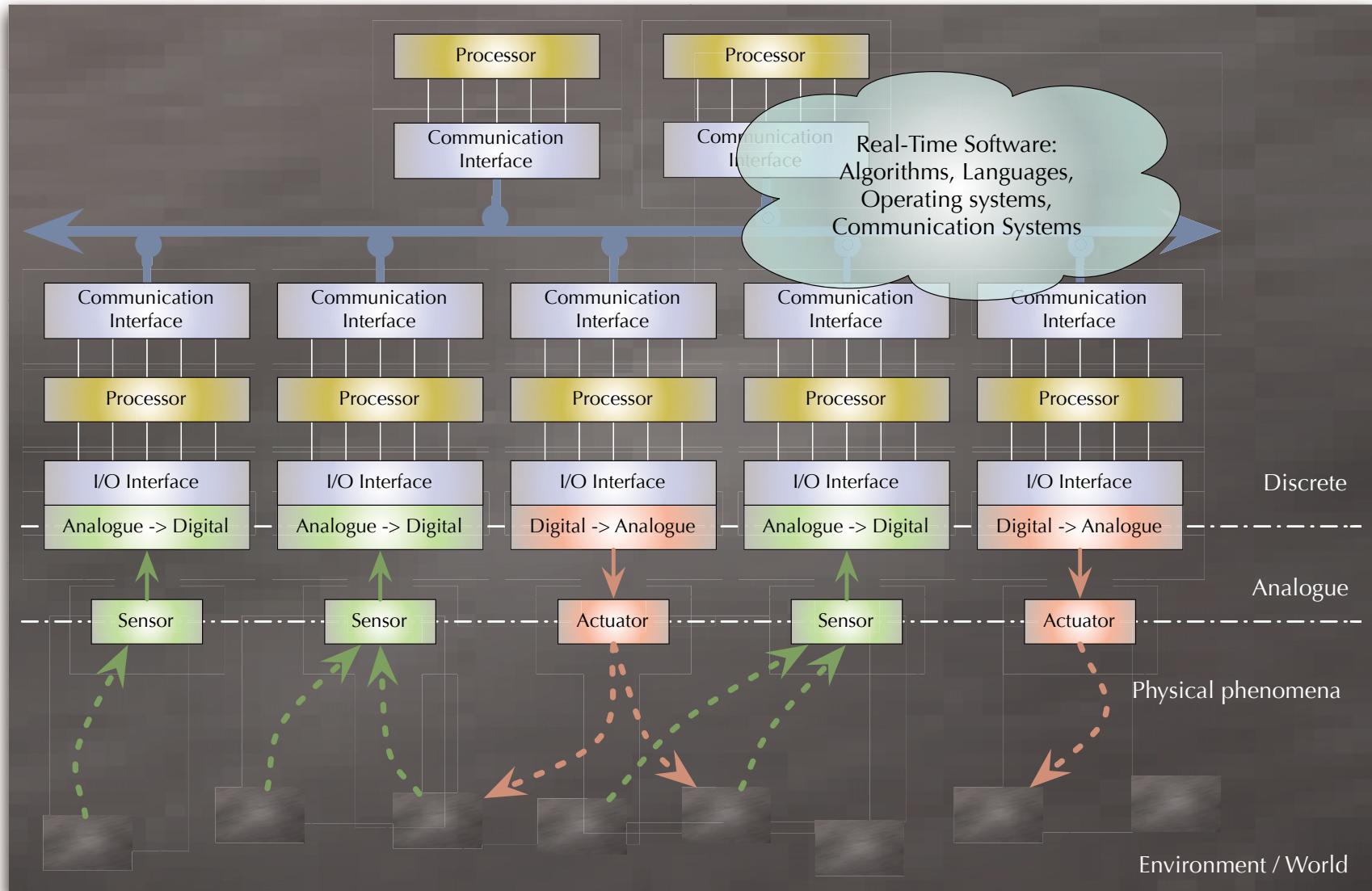
## Real-Time Systems Components





# Introduction & Languages

## Embedded Real-Time Systems Components





# *Introduction & Languages*

## *Real-Time Programming Languages*

### *Relevant Programming Paradigms*

- *Control flow*: **Imperative**  $\leftrightarrow$  **Declarative**
- *Declarative*: **Functional**  $\leftrightarrow$  **(Logic)**  $\leftrightarrow$  **Finite State Machines**
- *Allocations and bindings*: **Static**  $\leftrightarrow$  **Dynamic**
- *Time*: **Event-driven**  $\leftrightarrow$  **Discrete**  $\leftrightarrow$  **Synchronous**  $\leftrightarrow$  **Continuous**
- *Focus*: **Control flow-oriented**  $\leftrightarrow$  **Data flow-oriented**
- *Degree of concurrency*: **Sequential**  $\leftrightarrow$  **Concurrent**  $\leftrightarrow$  **Distributed**
- *Structure*: **Modular**  $\leftrightarrow$  **Generics**  $\leftrightarrow$  **Templates**  
 $\leftrightarrow$  **(Object-Oriented)**  $\leftrightarrow$  **(Aspect-Oriented)**  $\leftrightarrow$  **(Agent-Oriented)**
- *Determinism*: **Deterministic**  $\leftrightarrow$  **Non-deterministic**



# *Introduction & Languages*

## *Real-Time Programming Languages*

### *Requirements for Real-Time Languages / Environments*

- **Predictability**
  - ☞ No operations shall lead to unforeseeable timing behaviours.
- **Time**
  - ☞ Specified granularity, operations based on time, scheduling.
- **High integrity**
  - ☞ Complete, unambiguous language definition.
  - ☞ Strong compilers and runtime environments detecting faults as early as possible.
- **Concurrency and Distribution**
  - ☞ Solid, high-level synchronization and communication primitives, automated data marshallling.
- **Specific yet Scaling**
  - ☞ Mapping physical interfaces into high-level data-types and programming “in the very large”.



# *Introduction & Languages*

## *Real-Time Programming Languages*

### *Requirements for Real-Time Languages / Environments*

- **Predictability**
  - ☞ No operations shall lead to unforeseeable timing behaviours.
- **Time**
  - ☞ Specified granularity, operations based on time, scheduling.
- **High integrity**
  - ☞ Complete, unambiguous language definition.
  - ☞ Strong compilers and runtime environments detecting faults as early as possible.
- **Concurrency and Distribution**
  - ☞ Solid, high-level synchronization and communication primitives, automated data marshallling.
- **Specific yet Scalable**
  - ☞ Mapping physical interfaces into high-level data-types and programming "in the very large".



# *Introduction & Languages*

## *Real-Time Programming Languages*

# *Real-Time Languages, Operating Systems and Libraries*

What if you “cannot / want-not” use a real-time language and you need to formulate some/all real-time constraints outside the programming language?

### **Real-time operating systems:**

- ☞ Scheduling, interrupt handling, (potentially other features) migrate from the compiler environment into the operating system.
- ☞ Compiler level analysis is replaced by equivalent tools on the OS level.  
This requires additional languages, as those tools need specifications as well.

### **Libraries:**

- ☞ Loss of all compiler-level checks.
- ☞ Loss of all block structure and scoping.



# *Introduction & Languages*

## *Real-Time Programming Languages*

### *Some RT-Languages*

- Ada (Ada2012) ↗ General workhorse.
- Real-Time Java (Real-Time Specification for Java 1.1) ↗ Soft real-time applications.
- Esterel (Esterel v7) ↗ An alternative for high-integrity applications.
- VHDL ↗ Compile real-time data flows and independent, asynchronous control paths direct to hardware.
- Timed CSP (as used and developed since 1986) ↗ An algebraic approach.
- PEARL (PEARL-90) ↗ A traditional language, specialized on plant modelling.
- POSIX (POSIX 1003.1b, ...) ↗ The libraries of bare bone integers and semaphores.
- Assemblers / C ↗ The languages of bare bone words.



# *Introduction & Languages*

***Languages explicitly supporting concurrency: e.g. Ada***

Ada is an **ISO standardized** (ISO/IEC 8652:201x(E)) ‘general purpose’ language with focus on “program reliability and maintenance, programming as a human activity, and efficiency”.

It provides **core language primitives** for:

- Strong typing, contracts, separate compilation (specification and implementation), object-orientation.
- Concurrency, message passing, synchronization, monitors, rpcs, timeouts, scheduling, priority ceiling locks, hardware mappings, fully typed network communication.
- Strong run-time environments (incl. stand-alone execution).

... as well as **standardized language-annexes** for:

- Additional real-time features, distributed programming, system-level programming, numeric, information systems, safety and security issues.



# *Introduction & Languages*

## *Ada*

### *A crash course*

*... refreshing for some, x'th-language introduction for others:*

- **Specification** and **implementation** (body) parts, basic types
- **Exceptions**
- Information hiding in specifications ('**private**')
- **Contracts**
- **Generic** programming (polymorphism)
- **Tasking**
- Monitors and synchronisation ('**protected**', '**entries**', '**selects**', '**accepts**')
- Groups of interfaces ('**entry families**')
- **Abstract** types and **dispatching**

Not mentioned here: general object orientation, dynamic memory management, foreign language interfaces, marshalling, basics of imperative programming, ...

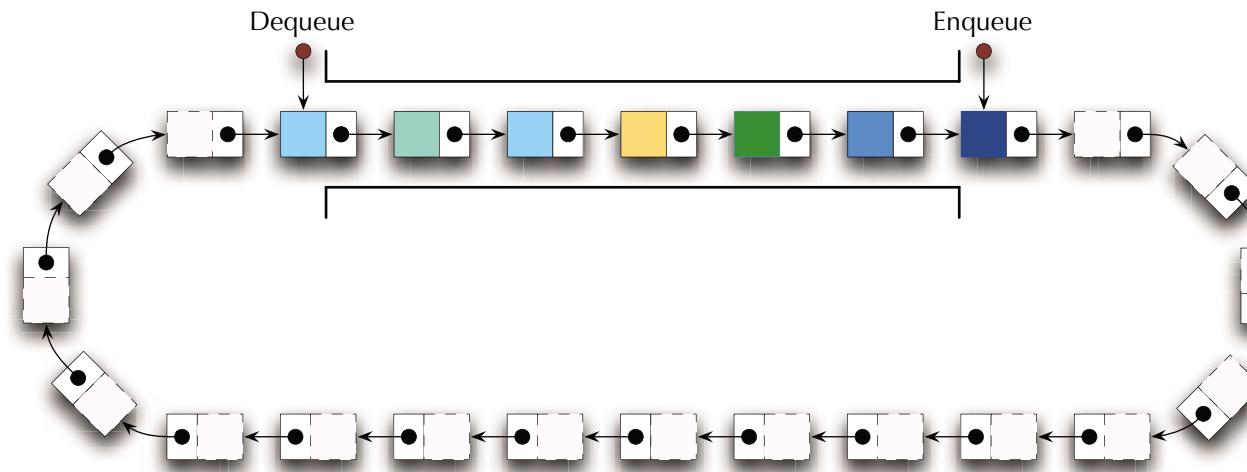
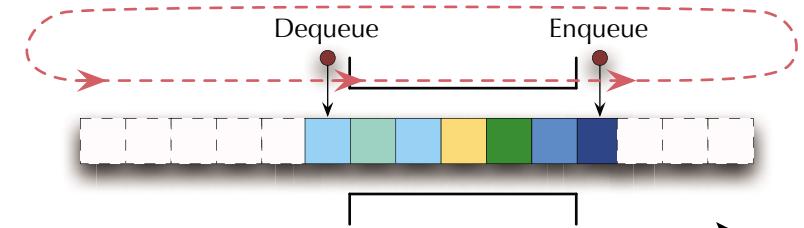
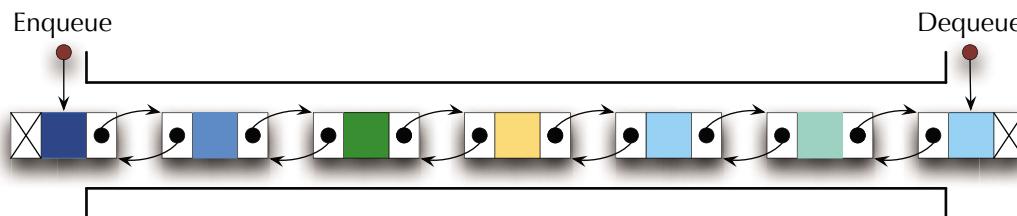
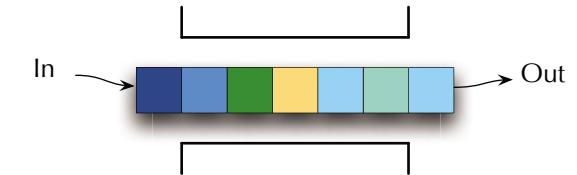


# Introduction & Languages

## Data structure example

### Queues

Forms of implementation:



Ring lists

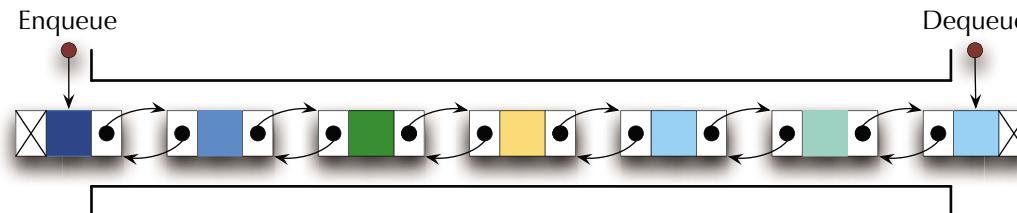
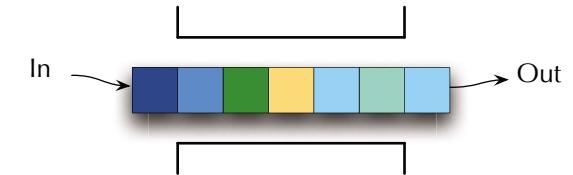


# Introduction & Languages

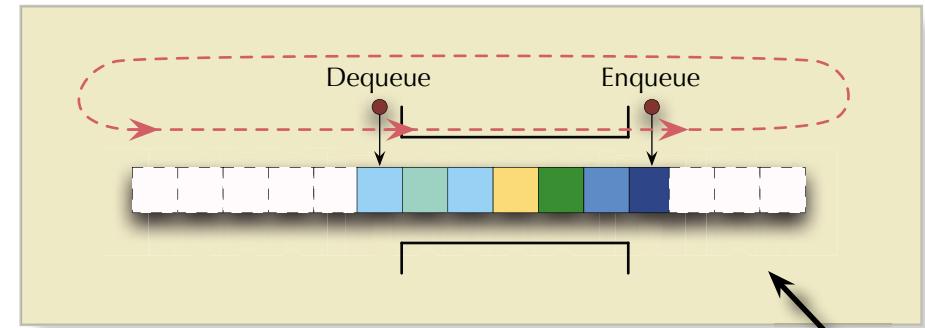
## Data structure example

### Queues

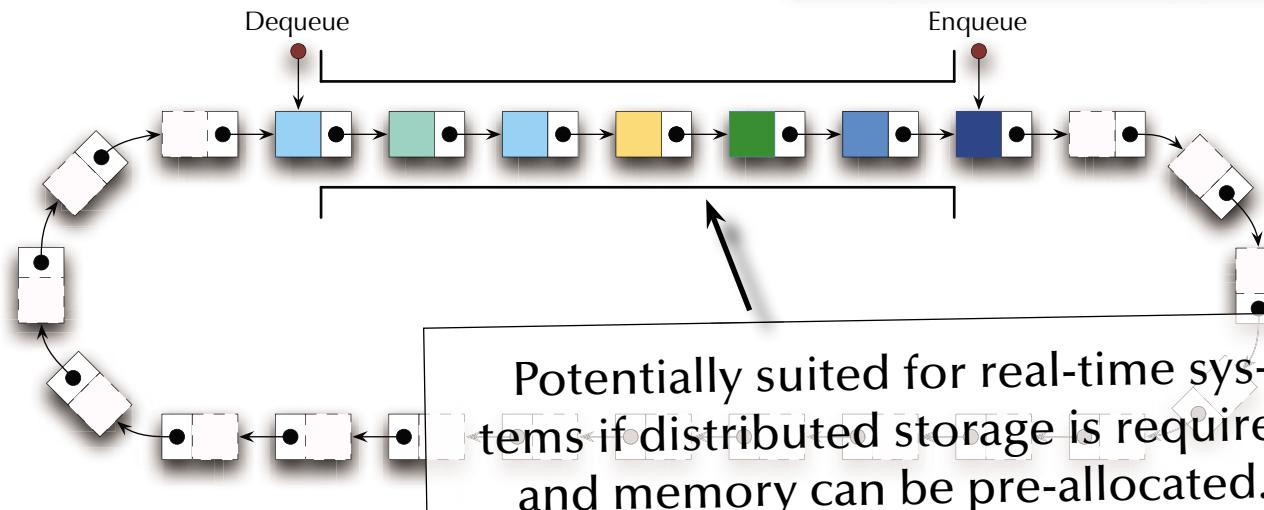
Forms of implementation:



Almost impossible for real-time systems.



Best suited for real-time systems.



Potentially suited for real-time systems if distributed storage is required and memory can be pre-allocated.



# *Introduction & Languages*

*Ada*

**Basics**

... introducing:

- **Specification and implementation** (body) parts
- **Constants**
- Some **basic types** (integer specifics)
- Some **type attributes**
- **Parameter specification**



# *Introduction & Languages*

## *A simple queue specification*

```
package Queue_Pack_Simple is
    QueueSize : constant Positive := 10;
    type Element      is new Positive range 1_000..40_000;
    type Marker      is mod QueueSize;
    type List         is array (Marker) of Element;
    type Queue_Type is record
        Top, Free : Marker := Marker'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;
    procedure Enqueue (Item: Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full   (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;
```



# Introduction & Languages

## A simple queue *specification*

```
package Queue_Pack_Simple is
```

```
    QueueSize : constant Positive := 10;
```

```
    type Element      is new Positive range 1_000..40_000;
```

```
    type Marker       is mod QueueSize;
```

```
    type List          is array (Marker) of Element;
```

```
    type Queue_Type   is record
```

```
        Top, Free : Marker := Marker'First;
```

```
        Is_Empty : Boolean := True;
```

```
        Elements : List;
```

```
    end record;
```

```
    procedure Enqueue (Item: Element; Queue: in out Queue_Type);
```

```
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
```

```
    function Is_Empty (Queue : Queue_Type) return Boolean;
```

```
    function Is_Full   (Queue : Queue_Type) return Boolean;
```

```
end Queue_Pack_Simple;
```

Specifications define an interface to provided types and operations.

Syntactically enclosed in a package block.



# Introduction & Languages

## A simple queue specification

```
package Queue_Pack_Simple is
    QueueSize : constant Positive := 10;
    type Element      is new Positive range 1_000..40_000;
    type Marker      is mod QueueSize;
    type List         is array (Marker) of Element;
    type Queue_Type  is record
        Top, Free : Marker := Marker'First;
        Is_Empty  : Boolean := True;
        Elements   : List;
    end record;
    procedure Enqueue (Item: Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full   (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;
```

Variables should be initialized.  
Constants must be initialized.

Assignments are denoted  
by the “:=” symbol.  
... leaving the “=” symbol  
for comparisons.



# Introduction & Languages

## A simple queue specification

```
package Queue_Pack_Simple is

    QueueSize : constant Positive := 10;

    type Element      is new Positive range 1_000..40_000;
    type Marker      is mod QueueSize;
    type List         is array (Marker) of Element;
    type Queue_Type is record
        Top, Free : Marker := Marker'First;
        Is_Empty : Boolean := True;
        Elements : List; ←
    end record;

    procedure Enqueue (Item: Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full   (Queue : Queue_Type) return Boolean;

end Queue_Pack_Simple;
```

Default initializations can  
be selected to be:

as is (random memory content),  
initialized to invalids, e.g. 999  
or valid, predictable values, e.g. 1\_000



# Introduction & Languages

## A simple queue *specification*

```
package Queue_Pack_Simple is
    QueueSize : constant Positive := 10;
    type Element      is new Positive range 1_000..40_000;
    type Marker      is mod QueueSize;
    type List         is array (Marker) of Element;
    type Queue_Type is record
        Top, Free : Marker := Marker'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;
    procedure Enqueue (Item: Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full   (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;
```

Numerical types  
can be specified by:

range, modulo,  
number of digits (☞ floating point)  
or delta increment (☞ fixed point).

Always be as specific as  
the language allows.  
... and don't repeat yourself!



# Introduction & Languages

## A simple queue *specification*

```
package Queue_Pack_Simple is
    QueueSize : constant Positive := 10;
    type Element      is new Positive range 1_000..40_000;
    type Marker      is mod QueueSize;
    type List         is array (Marker) of Element;
    type Queue_Type  is record
        Top, Free : Marker := Marker'First;
        Is_Empty  : Boolean := True;
        Elements   : List;
    end record;
    procedure Enqueue (Item:      Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full   (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;
```

All Types come with a long list of built-in attributes.

Let the compiler fill in what you already (implicitly) specified!



# Introduction & Languages

## A simple queue **specification**

```
package Queue_Pack_Simple is
    QueueSize : constant Positive := 10;
    type Element      is new Positive range 1_000..40_000;
    type Marker       is mod QueueSize;
    type List          is array (Marker) of Element;
    type Queue_Type   is record
        Top, Free : Marker := Marker'First;
        Is_Empty  : Boolean := True;
        Elements  : List;
    end record;
    procedure Enqueue (Item:      Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full   (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;
```

**Parameters** can be passed  
as '**in**' (default),  
**'out'**  
or '**in out**'.



# Introduction & Languages

## A simple queue *specification*

```
package Queue_Pack_Simple is
    QueueSize : constant Positive := 10;
    type Element      is new Positive range 1_000..40_000;
    type Marker      is mod QueueSize;
    type List         is array (Marker) of Element;
    type Queue_Type is record
        Top, Free : Marker := Marker'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;
    procedure Enqueue (Item: Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full   (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;
```

All specifications are used in  
*Code optimizations* (optional),  
*Compile time checks* (mandatory)  
*Run-time checks* (suppressible).



# Introduction & Languages

## A simple queue *specification*

```
package Queue_Pack_Simple is
    QueueSize : constant Positive := 10;
    type Element      is new Positive range 1_000..40_000;
    type Marker      is mod QueueSize;
    type List         is array (Marker) of Element;
    type Queue_Type is record
        Top, Free : Marker := Marker'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;
    procedure Enqueue (Item: Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full   (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;
```

... anything on this slide  
still not perfectly clear?



# *Introduction & Languages*

## *A simple queue **implementation***

```
package body Queue_Pack_Simple is
  procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
    begin
      Queue.Elements (Queue.Free) := Item;
      Queue.Free           := Queue.Free + 1;
      Queue.Is_Empty       := False;
    end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
    begin
      Item          := Queue.Elements (Queue.Top);
      Queue.Top     := Queue.Top + 1;
      Queue.Is_Empty := Queue.Top = Queue.Free;
    end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
    (Queue.Is_Empty);
  function Is_Full   (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Simple;
```



# Introduction & Languages

## A simple queue **implementation**

```
package body Queue_Pack_Simple is

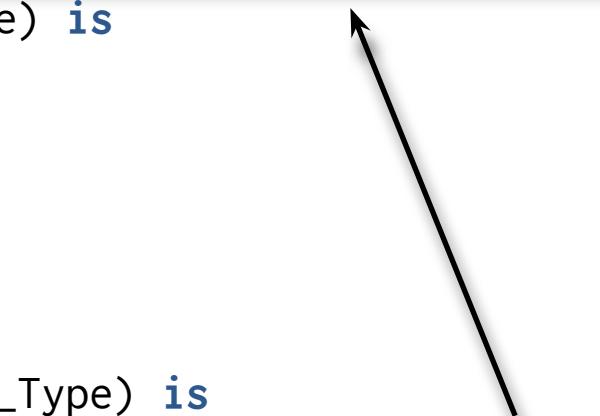
procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free           := Queue.Free + 1;
    Queue.Is_Empty       := False;
end Enqueue;

procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
begin
    Item          := Queue.Elements (Queue.Top);
    Queue.Top     := Queue.Top + 1;
    Queue.Is_Empty := Queue.Top = Queue.Free;
end Dequeue;

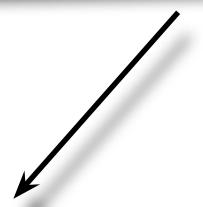
function Is_Empty (Queue : Queue_Type) return Boolean is
    (Queue.Is_Empty);

function Is_Full   (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);

end Queue_Pack_Simple;
```



Syntactically enclosed in a package body block.





# Introduction & Languages

## A simple queue **implementation**

```
package body Queue_Pack_Simple is
  procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
    begin
      Queue.Elements (Queue.Free) := Item;
      Queue.Free           := Queue.Free + 1;
      Queue.Is_Empty        := False;
    end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
    begin
      Item          := Queue.Elements (Queue.Top);
      Queue.Top     := Queue.Top + 1;
      Queue.Is_Empty := Queue.Top = Queue.Free;
    end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
    (Queue.Is_Empty);
  function Is_Full   (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Simple;
```

**Modulo type**, hence no index checks required.



# Introduction & Languages

## A simple queue **implementation**

```
package body Queue_Pack_Simple is
procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free            := Queue.Free + 1;
    Queue.Is_Empty        := False;
end Enqueue;
procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
begin
    Item           := Queue.Elements (Queue.Top);
    Queue.Top      := Queue.Top + 1;
    Queue.Is_Empty := Queue.Top = Queue.Free;
end Dequeue;
function Is_Empty (Queue : Queue_Type) return Boolean is
    (Queue.Is_Empty);
function Is_Full   (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Simple;
```

Boolean expressions



# Introduction & Languages

## A simple queue **implementation**

```
package body Queue_Pack_Simple is

procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free            := Queue.Free + 1;
    Queue.Is_Empty        := False;
end Enqueue;

procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
begin
    Item           := Queue.Elements (Queue.Top);
    Queue.Top      := Queue.Top + 1;
    Queue.Is_Empty := Queue.Top = Queue.Free;
end Dequeue;

function Is_Empty (Queue : Queue_Type) return Boolean is
    (Queue.Is_Empty);

function Is_Full   (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);

end Queue_Pack_Simple;
```

Side-effect free,  
**single expression functions**  
can be expressed with-  
out begin-end blocks.



# *Introduction & Languages*

## *A simple queue **implementation***

```
package body Queue_Pack_Simple is

procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free            := Queue.Free + 1;
    Queue.Is_Empty        := False;
end Enqueue;

procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
begin
    Item           := Queue.Elements (Queue.Top);
    Queue.Top      := Queue.Top + 1;
    Queue.Is_Empty := Queue.Top = Queue.Free;
end Dequeue;

function Is_Empty (Queue : Queue_Type) return Boolean is
    (Queue.Is_Empty);

function Is_Full   (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);

end Queue_Pack_Simple;
```

... anything on this slide  
still not perfectly clear?



# *Introduction & Languages*

## *A simple queue test program*

```
with Queue_Pack_Simple; use Queue_Pack_Simple;  
procedure Queue_Test_Simple is  
    Queue : Queue_Type;  
    Item   : Element;  
  
begin  
    Enqueue (2000, Queue);  
    Dequeue (Item, Queue);  
    Dequeue (Item, Queue);  
end Queue_Test_Simple;
```



# *Introduction & Languages*

## *A simple queue test program*

```
with Queue_Pack_Simple; use Queue_Pack_Simple;  
  
procedure Queue_Test_Simple is  
  
    Queue : Queue_Type;  
    Item   : Element;  
  
begin  
    Enqueue (2000, Queue);  
    Dequeue (Item, Queue);  
    Dequeue (Item, Queue);  
end Queue_Test_Simple;
```

Importing items from other packages  
is done with `with`-clauses.  
`use`-clauses allow to use names with  
qualifying them with the package name.



# *Introduction & Languages*

## *A simple queue test program*

```
with Queue_Pack_Simple; use Queue_Pack_Simple;  
procedure Queue_Test_Simple is  
    Queue : Queue_Type;  
    Item   : Element;  
  
begin  
    Enqueue (2000, Queue);  
    Dequeue (Item, Queue);  
    Dequeue (Item, Queue);  
end Queue_Test_Simple;
```

A top level procedure is read as the code which needs to be executed.



# *Introduction & Languages*

## *A simple queue test program*

```
with Queue_Pack_Simple; use Queue_Pack_Simple;  
procedure Queue_Test_Simple is  
    Queue : Queue_Type;  
    Item   : Element;  
  
begin  
    Enqueue (2000, Queue);  
    Dequeue (Item, Queue);  
    Dequeue (Item, Queue);  
end Queue_Test_Simple;
```

Variables are declared Algol style:  
“Item is of type Element”.



# *Introduction & Languages*

## *A simple queue test program*

```
with Queue_Pack_Simple; use Queue_Pack_Simple;  
procedure Queue_Test_Simple is  
    Queue : Queue_Type;  
    Item   : Element;  
  
begin  
    Enqueue (2000, Queue);  
    Dequeue (Item, Queue);  
    Dequeue (Item, Queue);  
end Queue_Test_Simple;
```

Will produce a result according to the chosen initialization:  
Raises an “invalid data” exception if initialized to invalids.

... hmm, ok ... so this was rubbish ...



# *Introduction & Languages*

## *A simple queue test program*

```
with Queue_Pack_Simple; use Queue_Pack_Simple;  
procedure Queue_Test_Simple is  
    Queue : Queue_Type;  
    Item   : Element;  
  
begin  
    Enqueue (2000, Queue);  
    Dequeue (Item, Queue);  
    Dequeue (Item, Queue);  
end Queue_Test_Simple;
```

... anything on this slide  
still not perfectly clear?



# *Introduction & Languages*

*Ada*

## *Exceptions*

... introducing:

- **Exception handling**
- **Enumeration types**
- **Type attributed operators**



# *Introduction & Languages*

## *A queue specification with proper exceptions*

```
package Queue_Pack_Exceptions is

    QueueSize : constant Positive := 10;

    type Element      is (Up, Down, Spin, Turn);
    type Marker       is mod QueueSize;
    type List          is array (Marker) of Element;

    type Queue_Type   is record
        Top, Free : Marker := Marker'First;
        Is_Empty  : Boolean := True;
        Elements   : List;
    end record;

    procedure Enqueue (Item:      Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
    function Is_Full   (Queue : Queue_Type) return Boolean is
        (not Queue.Is_Empty and then Queue.Top = Queue.Free);

    Queue_overflow, Queue_underflow : exception;

end Queue_Pack_Exceptions;
```



# Introduction & Languages

## A queue **specification** with proper exceptions

```
package Queue_Pack_Exceptions is

    QueueSize : constant Positive := 10;

    type Element      is (Up, Down, Spin, Turn);
    type Marker       is mod QueueSize;
    type List         is array (Marker) of Element;
    type Queue_Type   is record
        Top, Free : Marker := Marker'First;
        Is_Empty  : Boolean := True;
        Elements  : List;
    end record;

    procedure Enqueue (Item:      Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
    function Is_Full   (Queue : Queue_Type) return Boolean is
        (not Queue.Is_Empty and then Queue.Top = Queue.Free);

    Queue_overflow, Queue_underflow : exception;

end Queue_Pack_Exceptions;
```

Enumeration types are first-class types and can be used e.g. as array indices.

The representation values can be controlled and do not need to be continuous (e.g. for purposes like interfacing with hardware).



# Introduction & Languages

## A queue **specification** with proper exceptions

```
package Queue_Pack_Exceptions is

    QueueSize : constant Positive := 10;

    type Element      is (Up, Down, Spin, Turn);
    type Marker      is mod QueueSize;
    type List         is array (Marker) of Element;
    type Queue_Type   is record
        Top, Free : Marker := Marker'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;

    procedure Enqueue (Item:      Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
    function Is_Full   (Queue : Queue_Type) return Boolean is
        (not Queue.Is_Empty and then Queue.Top = Queue.Free);

    Queue_overflow, Queue_underflow : exception;
end Queue_Pack_Exceptions;
```

Nothing else changes  
in the specifications.

Exceptions need to be declared.



# Introduction & Languages

## A queue **specification** with proper exceptions

```
package Queue_Pack_Exceptions is

    QueueSize : constant Positive := 10;

    type Element      is (Up, Down, Spin, Turn);
    type Marker       is mod QueueSize;
    type List         is array (Marker) of Element;

    type Queue_Type   is record
        Top, Free : Marker := Marker'First;
        Is_Empty  : Boolean := True;
        Elements   : List;
    end record;

    procedure Enqueue (Item:      Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
    function Is_Full   (Queue : Queue_Type) return Boolean is
        (not Queue.Is_Empty and then Queue.Top = Queue.Free);

    Queue_overflow, Queue_underflow : exception;

end Queue_Pack_Exceptions;
```

... anything on this slide  
still not perfectly clear?

## A queue **implementation** with proper exceptions

```
package body Queue_Pack_Exceptions is
    procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
        begin
            if Is_Full (Queue) then
                raise Queue_overflow;
            end if;
            Queue.Elements (Queue.Free) := Item;
            Queue.Free      := Marker'Succ (Queue.Free);
            Queue.Is_Empty := False;
        end Enqueue;

    procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
        begin
            if Is_Empty (Queue) then
                raise Queue_underflow;
            end if;
            Item          := Queue.Elements (Queue.Top);
            Queue.Top      := Marker'Succ (Queue.Top);
            Queue.Is_Empty := Queue.Top = Queue.Free;
        end Dequeue;
    end Queue_Pack_Exceptions;
```

# A queue **implementation** with proper exceptions

```
package body Queue_Pack_Exceptions is
    procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
        begin
            if Is_Full (Queue) then
                raise Queue_overflow;
            end if;
            Queue.Elements (Queue.Free) := Item;
            Queue.Free      := Marker'Succ (Queue.Free);
            Queue.Is_Empty := False;
        end Enqueue;

        procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
        begin
            if Is_Empty (Queue) then
                raise Queue_underflow;
            end if;
            Item          := Queue.Elements (Queue.Top);
            Queue.Top      := Marker'Succ (Queue.Top);
            Queue.Is_Empty := Queue.Top = Queue.Free;
        end Dequeue;
    end Queue_Pack_Exceptions;
```

Raised **exceptions** break the control flow and “propagate” to the closest “exception handler” in the call-chain.

# A queue **implementation** with proper exceptions

```
package body Queue_Pack_Exceptions is
    procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
        begin
            if Is_Full (Queue) then
                raise Queue_overflow;
            end if;
            Queue.Elements (Queue.Free) := Item;
            Queue.Free      := Marker'Succ (Queue.Free);
            Queue.Is_Empty := False;
        end Enqueue;

        procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
        begin
            if Is_Empty (Queue) then
                raise Queue_underflow;
            end if;
            Item          := Queue.Elements (Queue.Top);
            Queue.Top      := Marker'Succ (Queue.Top);
            Queue.Is_Empty := Queue.Top = Queue.Free;
        end Dequeue;
    end Queue_Pack_Exceptions;
```

All Types come with a long list of built-in operators.  
Syntactically expressed as **attributes**.

Type attributes often make code more generic: ‘Succ works for instance on enumeration types as well ... “+ 1” does not.

# A queue **implementation** with proper exceptions

```
package body Queue_Pack_Exceptions is
    procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
        begin
            if Is_Full (Queue) then
                raise Queue_overflow;
            end if;
            Queue.Elements (Queue.Free) := Item;
            Queue.Free      := Marker'Succ (Queue.Free);
            Queue.Is_Empty := False;
        end Enqueue;

    procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
        begin
            if Is_Empty (Queue) then
                raise Queue_underflow;
            end if;
            Item          := Queue.Elements (Queue.Top);
            Queue.Top      := Marker'Succ (Queue.Top);
            Queue.Is_Empty := Queue.Top = Queue.Free;
        end Dequeue;
    end Queue_Pack_Exceptions;
```

... anything on this slide  
still not perfectly clear?



# *Introduction & Languages*

## *A queue test **program** with proper exceptions*

```
with Queue_Pack_Exceptions; use Queue_Pack_Exceptions;
with Ada.Text_IO          ; use Ada.Text_IO;

procedure Queue_Test_Exceptions is
    Queue : Queue_Type;
    Item   : Element;

begin
    Enqueue (Turn, Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue); -- will produce a Queue_underflow exception

exception
    when Queue_underflow => Put ("Queue underflow");
    when Queue_overflow   => Put ("Queue overflow");

end Queue_Test_Exceptions;
```



# Introduction & Languages

## A queue test **program** with proper exceptions

```
with Queue_Pack_Exceptions; use Queue_Pack_Exceptions;
with Ada.Text_IO          ; use Ada.Text_IO;

procedure Queue_Test_Exceptions is
    Queue : Queue_Type;
    Item   : Element;
begin
    Enqueue (Turn, Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue); -- will produce a Queue_underflow exception

exception
    when Queue_underflow => Put ("Queue underflow");
    when Queue_overflow   => Put ("Queue overflow");
end Queue_Test_Exceptions;
```

An **exception handler** has a choice to **handle**, **pass**, or **re-raise** the same or a different exception.

Raised **exceptions** break the control flow and “propagate” to the closest “exception handler” in the call-chain.

Control flow is continued after the **exception handler** in case of a handled exception.



# *Introduction & Languages*

## *A queue test **program** with proper exceptions*

```
with Queue_Pack_Exceptions; use Queue_Pack_Exceptions;
with Ada.Text_IO          ; use Ada.Text_IO;

procedure Queue_Test_Exceptions is
    Queue : Queue_Type;
    Item   : Element;

begin
    Enqueue (Turn, Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue); -- will produce a Queue_underflow exception

exception
    when Queue_underflow => Put ("Queue underflow");
    when Queue_overflow   => Put ("Queue overflow");

end Queue_Test_Exceptions;
```

... anything on this slide  
still not perfectly clear?



# Introduction & Languages

## A queue **specification** with proper exceptions

```
package Queue_Pack_Exceptions is

    QueueSize : constant Positive := 10;

    type Element      is (Up, Down, Spin, Turn);
    type Marker       is mod QueueSize;
    type List          is array (Marker) of Element;

    type Queue_Type   is record
        Top, Free : Marker := Marker'First;
        Is_Empty  : Boolean := True;
        Elements   : List;
    end record;

    procedure Enqueue (Item:      Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

    function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
    function Is_Full   (Queue : Queue_Type) return Boolean is
        (not Queue.Is_Empty and then Queue.Top = Queue.Free);

    Queue_overflow, Queue_underflow : exception;

end Queue_Pack_Exceptions;
```

This package provides access to 'internal' structures which can lead to inconsistent access.



# *Introduction & Languages*

*Ada*

## *Information hiding*

... introducing:

- **Private declarations**  
☞ needed to compile specifications,  
yet not accessible for a user of the package.
- **Private types** ☞ assignments and comparisons are allowed
- **Limited private types** ☞ entity cannot be assigned or compared



# *Introduction & Languages*

## *A queue specification with proper information hiding*

```
package Queue_Pack_Private is

    QueueSize : constant Integer := 10;

    type Element is new Positive range 1..1000;
    type Queue_Type is limited private;

    procedure Enqueue (Item: Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full   (Queue : Queue_Type) return Boolean;

    Queueoverflow, Queueunderflow : exception;

private

    type Marker is mod QueueSize;
    type List is array (Marker) of Element;
    type Queue_Type is record
        Top, Free : Marker := Marker'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;

end Queue_Pack_Private;
```



# Introduction & Languages

## A queue **specification** with proper information hiding

```
package Queue_Pack_Private is
```

```
    QueueSize : constant Integer := 10;  
  
    type Element is new Positive range 1..1000;  
    type Queue_Type is limited private;  
  
    procedure Enqueue (Item: Element; Queue: in out Queue_Type);  
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);  
    function Is_Empty (Queue : Queue_Type) return Boolean;  
    function Is_Full (Queue : Queue_Type) return Boolean;  
  
    Queueoverflow, Queueunderflow : exception;
```

```
private
```

```
    type Marker is mod QueueSize;  
    type List is array (Marker) of Element;  
    type Queue_Type is record  
        Top, Free : Marker := Marker'First;  
        Is_Empty : Boolean := True;  
        Elements : List;  
    end record;
```

```
end Queue_Pack_Private;
```

private splits the specification into a public and a private section.

The private section is only here so that the specifications can be separately compiled.



# Introduction & Languages

## A queue **specification** with proper information hiding

```
package Queue_Pack_Private is
    QueueSize : constant Integer := 10;
    type Element is new Positive range 1..1000;
    type Queue_Type is limited private;
    procedure Enqueue (Item: Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full   (Queue : Queue_Type) return Boolean;
    Queueoverflow, Queueunderflow : exception;
private
    type Marker is mod QueueSize;
    type List is array (Marker) of Element;
    type Queue_Type is record
        Top, Free : Marker := Marker'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;
end Queue_Pack_Private;
```

Queue\_Type can now be used outside this package **without any way to access its internal structure.**

**limited disables assignments and comparisons** for this type.

A user of this package would now e.g. not be able to make a copy of a Queue\_Type value.



# Introduction & Languages

## A queue **specification** with proper information hiding

```
package Queue_Pack_Private is
    QueueSize : constant Integer := 10;
    type Element is new Positive range 1..1000;
    type Queue_Type is limited private;
    procedure Enqueue (Item: Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full   (Queue : Queue_Type) return Boolean;
    Queueoverflow, Queueunderflow : exception;
private
    type Marker is mod QueueSize;
    type List is array (Marker) of Element;
    type Queue_Type is record
        Top, Free : Marker := Marker'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;
end Queue_Pack_Private;
```

Queue\_Type can now be used outside this package **without any way to access its internal structure.**

Alternatively '=' and ':=' operations can be replaced with type-specific versions (overloaded) or default operations can be allowed.



# Introduction & Languages

## A queue **specification** with proper information hiding

```
package Queue_Pack_Private is

    QueueSize : constant Integer := 10;

    type Element is new Positive range 1..1000;
    type Queue_Type is limited private;

    procedure Enqueue (Item: Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full   (Queue : Queue_Type) return Boolean;

    Queueoverflow, Queueunderflow : exception;

private

    type Marker is mod QueueSize;
    type List is array (Marker) of Element;
    type Queue_Type is record
        Top, Free : Marker := Marker'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;

end Queue_Pack_Private;
```

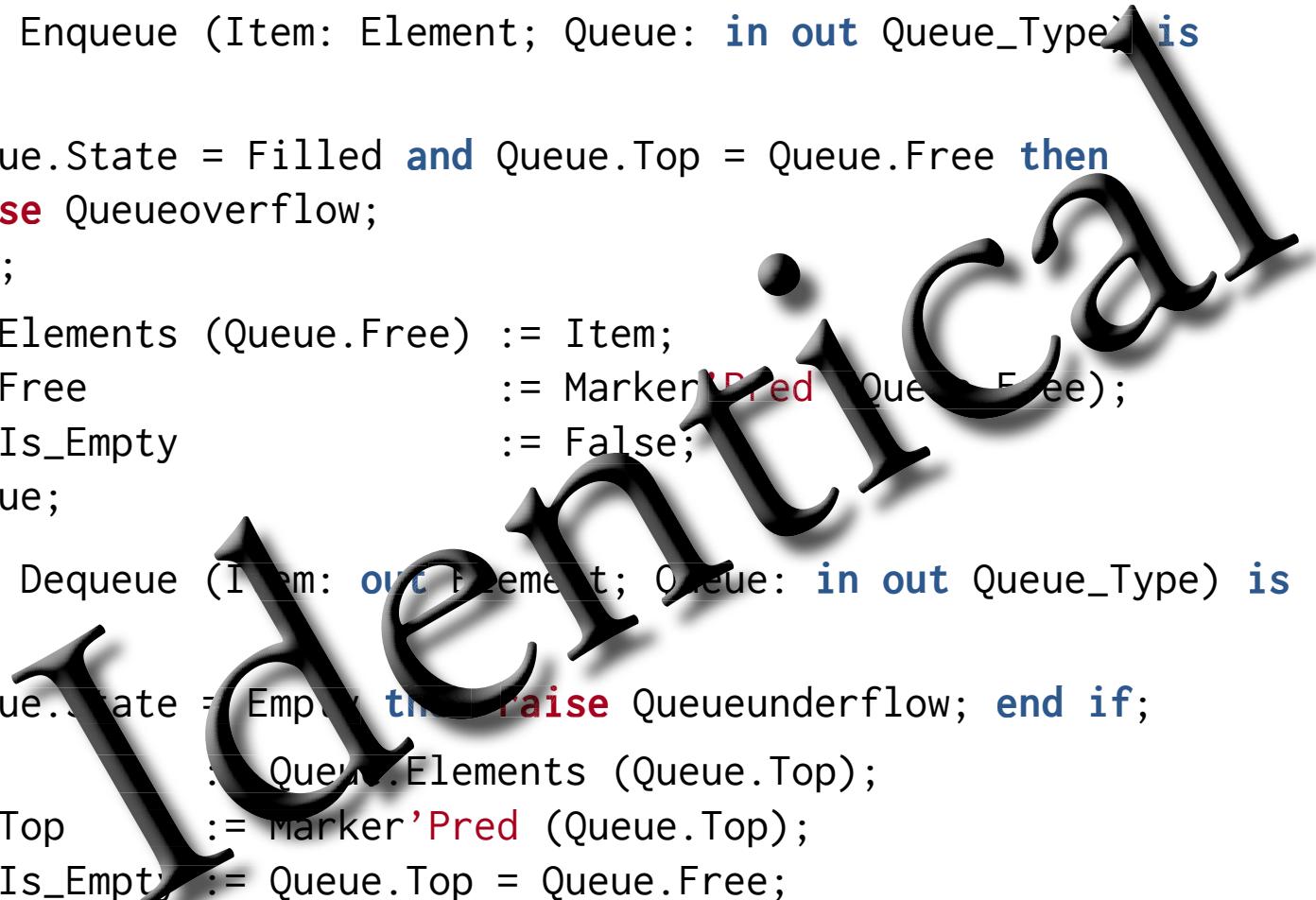
... anything on this slide  
still not perfectly clear?

# *A queue **implementation** with proper information hiding*

```
package body Queue_Pack_Private is
procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
        raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker'Pred (Queue.Free);
    Queue.Is_Empty := False;
end Enqueue;

procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
begin
    if Queue.State = Empty then raise Queueunderflow; end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker'Pred (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
end Dequeue;

function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Private;
```



# *A queue **implementation** with proper information hiding*

```
package body Queue_Pack_Private is
procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
begin
  if Queue.State = Filled and Queue.Top = Queue.Free then
    raise Queueoverflow;
  end if;
  Queue.Elements (Queue.Free) := Item;
  Queue.Free := Marker'Pred (Queue.Free);
  Queue.IsEmpty := False;
end Enqueue;

procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
begin
  if Queue.State = Empty then raise Queueunderflow; end if;
  Item := Queue.Elements (Queue.Top);
  Queue.Top := Marker'Pred (Queue.Top);
  Queue.IsEmpty := Queue.Top = Queue.Free;
end Dequeue;

function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.IsEmpty);
function Is_Full (Queue : Queue_Type) return Boolean is
  (not Queue.IsEmpty and then Queue.Top = Queue.Free);

end Queue_Pack_Private;
```

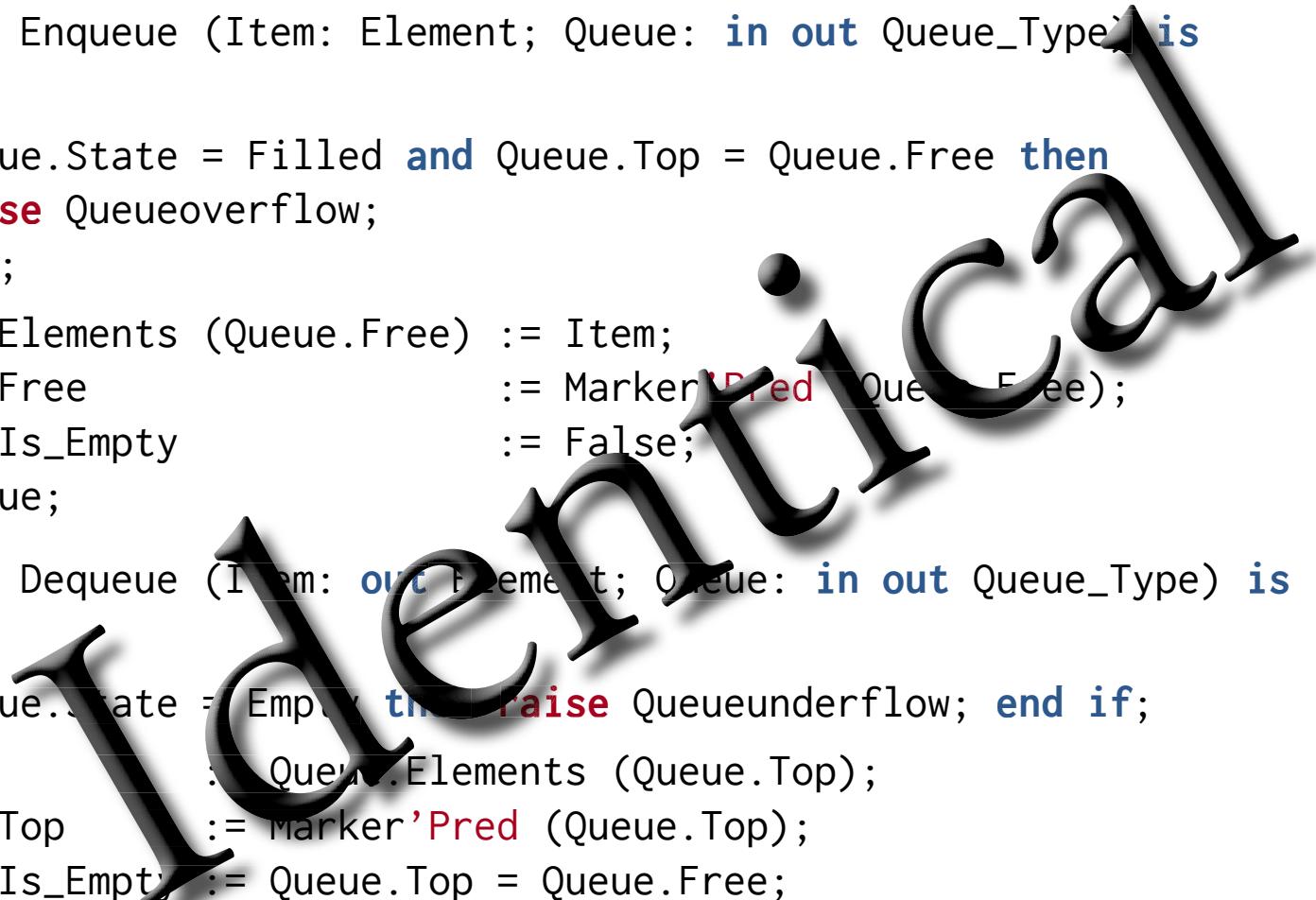
... besides the implementation of the two functions which has been moved to the implementation section.

# *A queue **implementation** with proper information hiding*

```
package body Queue_Pack_Private is
procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
        raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker'Pred (Queue.Free);
    Queue.Is_Empty := False;
end Enqueue;

procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
begin
    if Queue.State = Empty then raise Queueunderflow; end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker'Pred (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
end Dequeue;

function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Private;
```



... anything on this slide  
still not perfectly clear?



# *Introduction & Languages*

## *A queue test **program** with proper information hiding*

```
with Queue_Pack_Private; use Queue_Pack_Private;
with Ada.Text_IO          ; use Ada.Text_IO;

procedure Queue_Test_Private is
    Queue, Queue_Copy : Queue_Type;
    Item              : Element;

begin
    Queue_Copy := Queue;
    -- compiler-error: "left hand of assignment must not be limited type"
    Enqueue (Item => 1, Queue => Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue); -- would produce a "Queue underflow"

exception
    when Queueunderflow => Put ("Queue underflow");
    when Queueoverflow   => Put ("Queue overflow");
end Queue_Test_Private;
```



# Introduction & Languages

## A queue test **program** with proper information hiding

```
with Queue_Pack_Private; use Queue_Pack_Private;
with Ada.Text_IO          ; use Ada.Text_IO;

procedure Queue_Test_Private is
    Queue, Queue_Copy : Queue_Type;
    Item              : Element;
begin
    Queue_Copy := Queue;
    -- compiler-error: "left hand of assignment must not be limited type"
    Enqueue (Item => 1, Queue => Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue); -- would produce a "Queue underflow"

exception
    when Queueunderflow => Put ("Queue underflow");
    when Queueoverflow  => Put ("Queue overflow");
end Queue_Test_Private;
```

Illegal operation on a limited type.



# Introduction & Languages

## A queue test **program** with proper information hiding

```
with Queue_Pack_Private; use Queue_Pack_Private;
with Ada.Text_IO          ; use Ada.Text_IO;

procedure Queue_Test_Private is
    Queue, Queue_Copy : Queue_Type;
    Item              : Element;

begin
    Queue_Copy := Queue;
    -- compiler-error: "left hand of assignment must not be limited type"
    Enqueue (Item => 1, Queue => Queue); ←
    Dequeue (Item, Queue);
    Dequeue (Item, Queue); -- would produce a "Queue underflow"

exception
    when Queueunderflow => Put ("Queue underflow");
    when Queueoverflow  => Put ("Queue overflow");
end Queue_Test_Private;
```

Parameters can be named or passed by order of definition.  
(Named parameters do not need to follow the definition order.)



# Introduction & Languages

## A queue test **program** with proper information hiding

```
with Queue_Pack_Private; use Queue_Pack_Private;
with Ada.Text_IO          ; use Ada.Text_IO;

procedure Queue_Test_Private is
    Queue, Queue_Copy : Queue_Type;
    Item              : Element;

begin
    Queue_Copy := Queue;
    -- compiler-error: "left hand of assignment must not be limited type"
    Enqueue (Item => 1, Queue => Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue); -- would produce a "Queue underflow"

exception
    when Queueunderflow => Put ("Queue underflow");
    when Queueoverflow   => Put ("Queue overflow");
end Queue_Test_Private;
```

... anything on this slide  
still not perfectly clear?



# *Introduction & Languages*

## *Ada Contracts*

... introducing:

- **Pre- and Post-Conditions** on methods
- **Invariants** on types
- **For all, For any** predicates



# Introduction & Languages

## A contracting queue specification

```
package Queue_Pack_Contract is
    Queue_Size : constant Positive := 10;
    type Element is new Positive range 1 .. 1000;
    type Queue_Type is private;
    procedure Enqueue (Item : Element; Q : in out Queue_Type) with
        Pre => not Is_Full (Q),
        Post => not Is_Empty (Q) and then Length (Q) = Length (Q'Old) + 1
            and then Lookahead (Q, Length (Q)) = Item
            and then (for all ix in 1 .. Length (Q'Old)
                        => Lookahead (Q, ix) = Lookahead (Q'Old, ix));
    procedure Dequeue (Item : out Element; Q : in out Queue_Type) with
        Pre => not Is_Empty (Q),
        Post => not Is_Full (Q) and then Length (Q) = Length (Q'Old) - 1
            and then (for all ix in 1 .. Length (Q)
                        => Lookahead (Q, ix) = Lookahead (Q'Old, ix + 1));
    function Is_Empty (Q : Queue_Type) return Boolean;
    function Is_Full   (Q : Queue_Type) return Boolean;
    function Length   (Q : Queue_Type) return Natural;
    function Lookahead (Q : Queue_Type; Depth : Positive) return Element;
```



# Introduction & Languages

## A contracting queue specification

```
package Queue_Pack_Contract is
    Queue_Size : constant Positive := 10;
    type Element is new Positive range 1 .. 1000;
    type Queue_Type is private;

    procedure Enqueue (Item : Element; Q : in out Queue_Type) with
        Pre => not Is_Full (Q),
        Post => not Is_Empty (Q) and then Length (Q) = Length (Q'Old) + 1
                  and then Lookahead (Q, Length (Q)) = Item
                  and then (for all ix in 1 .. Length (Q'Old)
                             => Lookahead (Q, ix) = Lookahead (Q'Old, ix));
    procedure Dequeue (Item : out Element; Q : in out Queue_Type) with
        Pre => not Is_Empty (Q),
        Post => not Is_Full (Q) and then Length (Q) = Length (Q'Old) - 1
                  and then (for all ix in 1 .. Length (Q)
                             => Lookahead (Q, ix) = Lookahead (Q'Old, ix + 1));

    function Is_Empty (Q : Queue_Type) return Boolean;
    function Is_Full   (Q : Queue_Type) return Boolean;
    function Length   (Q : Queue_Type) return Natural;
    function Lookahead (Q : Queue_Type; Depth : Positive) return Element;
```

Pre- and Post-predicates are checked before and after each execution resp.

Original (Pre) values can still be referred to.



# Introduction & Languages

## A contracting queue specification

```
package Queue_Pack_Contract is
    Queue_Size : constant Positive := 10;
    type Element is new Positive range 1 .. 1000;
    type Queue_Type is private;

    procedure Enqueue (Item : Element; Q : in out Queue_Type) with
        Pre => not Is_Full (Q),
        Post => not Is_Empty (Q) and then Length (Q) = Length (Q'Old) + 1
                    and then Lookahead (Q, Length (Q)) = Item
                    and then (for all ix in 1 .. Length (Q'Old)
                                => Lookahead (Q, ix) = Lookahead (Q'Old, ix));

    procedure Dequeue (Item : out Element; Q : in out Queue_Type) with
        Pre => not Is_Empty (Q),
        Post => not Is_Full (Q) and then Length (Q) = Length (Q'Old) - 1
                    and then (for all ix in 1 .. Length (Q)
                                => Lookahead (Q, ix) = Lookahead (Q'Old, ix + 1));

    function Is_Empty (Q : Queue_Type) return Boolean;
    function Is_Full   (Q : Queue_Type) return Boolean;
    function Length   (Q : Queue_Type) return Natural;
    function Lookahead (Q : Queue_Type; Depth : Positive) return Element;
```

... anything on this slide  
still not perfectly clear?



# Introduction & Languages

## A contracting queue specification (cont.)

```
private
  type Marker is mod Queue_Size;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List; -- will be initialized to invalids
  end record with Type_Invariant
    => (not Queue_Type.Is_Empty or else Queue_Type.Top = Queue_Type.Free)
        and then (for all ix in 1 .. Length (Queue_Type)
                    => Lookahead (Queue_Type, ix)'Valid);
  function Is_Empty (Q : Queue_Type) return Boolean is (Q.Is_Empty);
  function Is_Full (Q : Queue_Type) return Boolean is
    (not Q.Is_Empty and then Q.Top = Q.Free);
  function Length (Q : Queue_Type) return Natural is
    (if Is_Full (Q) then Queue_Size else Natural (Q.Free - Q.Top));
  function Lookahead (Q : Queue_Type; Depth : Positive) return Element is
    (Q.Elements (Q.Top + Marker (Depth - 1)));
end Queue_Pack_Contract;
```



# Introduction & Languages

## A contracting queue specification (cont.)

private

```
type Marker is mod Queue_Size;
type List is array (Marker) of Element;
type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List; -- will be initialized to invalids
end record with Type_Invariant
```

```
=> (not Queue_Type.Is_Empty or else Queue_Type.Top = Queue_Type.Free)
    and then (for all ix in 1 .. Length (Queue_Type)
                => Lookahead (Queue_Type, ix)'Valid);
```

```
function Is_Empty (Q : Queue_Type) return Boolean is (Q.Is_Empty);
function Is_Full (Q : Queue_Type) return Boolean is
    (not Q.Is_Empty and then Q.Top = Q.Free);
function Length (Q : Queue_Type) return Natural is
    (if Is_Full (Q) then Queue_Size else Natural (Q.Free - Q.Top));
function Lookahead (Q : Queue_Type; Depth : Positive) return Element is
    (Q.Elements (Q.Top + Marker (Depth - 1)));
end Queue_Pack_Contract;
```

Type-Invariants are checked on return from any operation defined in the public part.



# Introduction & Languages

## A contracting queue specification (cont.)

private

```
type Marker is mod Queue_Size;
type List is array (Marker) of Element;
type Queue_Type is record
```

```
    Top, Free : Marker := Marker'First;
```

```
    Is_Empty : Boolean := True;
```

```
    Elements : List; -- will be initialized to invalids
```

```
end record with Type_Invariant
```

```
=> (not Queue_Type.Is_Empty or else Queue_Type.Top = Queue_Type.Free)
    and then (for all ix in 1 .. Length (Queue_Type)
```

```
        => Lookahead (Queue_Type, ix)'Valid);
```

```
function Is_Empty (Q : Queue_Type) return Boolean is (Q.Is_Empty);
```

```
function Is_Full (Q : Queue_Type) return Boolean is
    (not Q.Is_Empty and then Q.Top = Q.Free);
```

```
function Length (Q : Queue_Type) return Natural is
    (if Is_Full (Q) then Queue_Size else Natural (Q.Free - Q.Top));
```

```
function Lookahead (Q : Queue_Type; Depth : Positive) return Element is
    (Q.Elements (Q.Top + Marker (Depth - 1)));
```

```
end Queue_Pack_Contract;
```

... anything on this slide  
still not perfectly clear?



# *Introduction & Languages*

## *A contracting queue **implementation***

```
package body Queue_Pack_Contract is

procedure Enqueue (Item : Element; Q : in out Queue_Type) is
begin
    Q.Elements (Q.Free) := Item;
    Q.Free            := Q.Free + 1;
    Q.Is_Empty        := False;
end Enqueue;

procedure Dequeue (Item : out Element; Q : in out Queue_Type) is
begin
    Item      := Q.Elements (Q.Top);
    Q.Top    := Q.Top + 1;
    Q.Is_Empty = Q.Top = Q.Free;
end Dequeue;

end Queue_Pack_Contract;
```

No checks in the implementation part,  
as all required conditions have been  
guaranteed via the specifications.



# *Introduction & Languages*

## *A contracting queue test program*

```
with Ada.Text_IO;           use Ada.Text_IO;
with Exceptions;          use Exceptions;
with Queue_Pack_Contract; use Queue_Pack_Contract;
with System.Assertions;    use System.Assertions;

procedure Queue_Test_Contract is
    Queue : Queue_Type;
    Item   : Element;

begin
    Enqueue (Item => 1, Q => Queue);
    Enqueue (Item => 2, Q => Queue);
    Dequeue (Item, Queue); Put (Element'Image (Item));
    Dequeue (Item, Queue); Put (Element'Image (Item));
    Dequeue (Item, Queue); -- will produce an Assert_Failure
    Put (Element'Image (Item));
    Put ("Queue is empty on exit: "); Put (Boolean'Image (IsEmpty (Queue)));
exception
    when Exception_Id : Assert_Failure => Show_Exception (Exception_Id);
end Queue_Test_Contract;
```



# Introduction & Languages

## A *contracting queue test program*

```
with Ada.Text_IO;           use Ada.Text_IO;
with Exceptions;          use Exceptions;
with Queue_Pack_Contract; use Queue_Pack_Contract;
with System.Assertions;    use System.Assertions;

procedure Queue_Test_Contract is
    Queue : Queue_Type;
    Item   : Element;
begin
    Enqueue (Item => 1, Q => Queue);
    Enqueue (Item => 2, Q => Queue);
    Dequeue (Item, Queue); Put (Element'Image (Item));
    Dequeue (Item, Queue); Put (Element'Image (Item));
    Dequeue (Item, Queue); -- will produce an Assert_Failure
    Put (Element'Image (Item));
    Put ("Queue is empty on exit: "); Put (Boolean'Image (IsEmpty (Queue)));
exception
    when Exception_Id : Assert_Failure => Show_Exception (Exception_Id);
end Queue_Test_Contract;
```

**Violated Pre-condition** will raise  
an assert failure exception.



# *Introduction & Languages*

## *A contracting queue test program*

```
with Ada.Text_IO;           use Ada.Text_IO;
with Exceptions;          use Exceptions;
with Queue_Pack_Contract; use Queue_Pack_Contract;
with System.Assertions;    use System.Assertions;

procedure Queue_Test_Contract is
    Queue : Queue_Type;
    Item   : Element;

begin
    Enqueue (Item => 1, Q => Queue);
    Enqueue (Item => 2, Q => Queue);
    Dequeue (Item, Queue); Put (Element'Image (Item));
    Dequeue (Item, Queue); Put (Element'Image (Item));
    Dequeue (Item, Queue); -- will produce an Assert_Failure
    Put (Element'Image (Item));
    Put ("Queue is empty on exit: "); Put (Boolean'Image (IsEmpty (Queue)));

exception
    when Exception_Id : Assert_Failure => Show_Exception (Exception_Id);

end Queue_Test_Contract;
```

... anything on this slide  
still not perfectly clear?

**A contracted queue specification**

Exceptions are commonly preferred to handle rare, yet valid situations.

Contracts are commonly used to test program correctness with respect to its specifications.

```

package Queue_Pack_Contract is
  (...)

  procedure Enqueue (Item : Element; Q : in out Queue_Type) with
    Pre => not Is_Full (Q), -- could also be “=> True” according to specifications
    Post => not Is_Empty (Q) and then Length (Q) = Length (Q'Old) + 1
      and then Lookahead (Q, Length (Q)) = Item
      and then (for all ix in 1 .. Length (Q'Old)
        => Lookahead (Q, ix) = Lookahead (Q'Old, ix));
  
```

```

  procedure Dequeue (Item : out Element; Q : in out Queue_Type) with
    Pre => not Is_Empty (Q), -- could also be “=> True” according to specifications
    Post => not Is_Full (Q) and then Length (Q) = Length (Q'Old) - 1
      and then (for all ix in 1 .. Length (Q)
        => Lookahead (Q, ix) = Lookahead (Q'Old, ix + 1));
  
```

```

  (...)

  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record with Type_Invariant =>
    (not Queue_Type.Is_Empty or else Queue_Type.Top = Queue_Type.Free)
    and then (for all ix in 1 .. Length (Queue_Type)
      => Lookahead (Queue_Type, ix)'Valid);
  
```

```

  (...)
```

Those contracts can be used to fully specify operations and types. Specifications should be complete, consistent and canonical, while using as little implementation details as possible.



# *Introduction & Languages*

*Ada*

## *Generic (polymorphic) packages*

... introducing:

- Specification of **generic** packages
- Instantiation of **generic** packages



# *Introduction & Languages*

## *A generic queue specification*

```
generic
  type Element is private;
package Queue_Pack_Generic is
  QueueSize: constant Integer := 10;
  type Queue_Type is limited private;
  procedure Enqueue (Item:      Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full   (Queue : Queue_Type) return Boolean;
  Queueoverflow, Queueunderflow : exception;
private
  type Marker is mod QueueSize;
  type List  is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Generic;
```



# Introduction & Languages

## A generic queue *specification*

```
generic
  type Element is private;
package Queue_Pack_Generic is
  QueueSize: constant Integer := 10;
  type Queue_Type is limited private;
  procedure Enqueue (Item:     Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full   (Queue : Queue_Type) return Boolean;
  Queueoverflow, Queueunderflow : exception;
private
  type Marker is mod QueueSize;
  type List  is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Generic;
```

The type of Element now becomes a parameter of a generic package.

No restrictions (private) have been set for the type of Element.

Haskell syntax:  
enqueue :: a -> Queue a -> Queue a



# Introduction & Languages

## A generic queue specification

```
generic
  type Element is private;

package Queue_Pack_Generic is
  QueueSize: constant Integer := 10;
  type Queue_Type is limited private;
  procedure Enqueue (Item: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  Queueoverflow, Queueunderflow : exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Generic;
```

Generic aspects can include:

- Type categories
- Incomplete types
- Constants
- Procedures and functions
- Other packages
- Objects (interfaces)

Default values can be provided  
(making those parameters optional)



# *Introduction & Languages*

## *A generic queue specification*

```
generic
  type Element is private;
package Queue_Pack_Generic is
  QueueSize: constant Integer := 10;
  type Queue_Type is limited private;
  procedure Enqueue (Item:      Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full   (Queue : Queue_Type) return Boolean;
  Queueoverflow, Queueunderflow : exception;
private
  type Marker is mod QueueSize;
  type List  is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Generic;
```

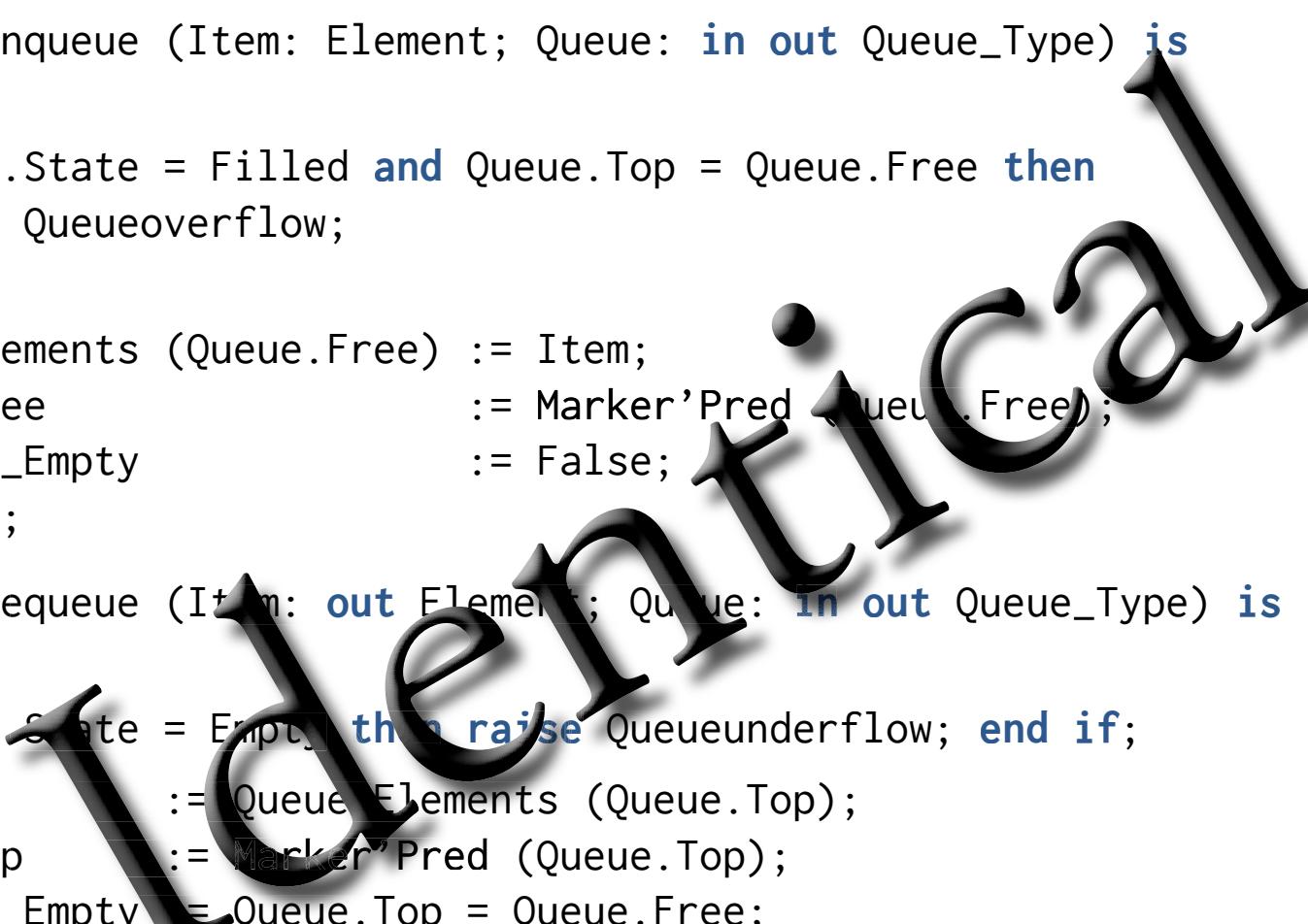
... anything on this slide  
still not perfectly clear?

# A generic queue implementation

```
package body Queue_Pack_Generic is
    procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
        begin
            if Queue.State = Filled and Queue.Top = Queue.Free then
                raise Queueoverflow;
            end if;
            Queue.Elements (Queue.Free) := Item;
            Queue.Free := Marker'Pred (Queue.Free);
            Queue.Is_Empty := False;
        end Enqueue;

        procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
        begin
            if Queue.State = Empty then raise Queueunderflow; end if;
            Item := Queue.Elements (Queue.Top);
            Queue.Top := Marker'Pred (Queue.Top);
            Queue.Is_Empty := Queue.Top = Queue.Free;
        end Dequeue;

        function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
        function Is_Full (Queue : Queue_Type) return Boolean is
            (not Queue.Is_Empty and then Queue.Top = Queue.Free);
    end Queue_Pack_Generic;
```





# *Introduction & Languages*

## *A generic queue test program*

```
with Queue_Pack_Generic; -- cannot apply 'use' clause here
with Ada.Text_IO          ; use Ada.Text_IO;
procedure Queue_Test_Generic is
  package Queue_Pack_Positive is
    new Queue_Pack_Generic (Element => Positive);
  use Queue_Pack_Positive; -- 'use' clause can be applied to instantiated package
  Queue : Queue_Type;
  Item   : Positive;
begin
  Enqueue (Item => 1, Queue => Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue); -- will produce a "Queue underflow"
exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow  => Put ("Queue overflow");
end Queue_Test_Generic;
```



# Introduction & Languages

## A generic queue test program

```
with Queue_Pack_Generic; -- cannot apply 'use' clause here
with Ada.Text_IO          ; use Ada.Text_IO;

procedure Queue_Test_Generic is
  package Queue_Pack_Positive is
    new Queue_Pack_Generic (Element => Positive);
  use Queue_Pack_Positive; -- 'use' clause can be applied to instantiated package
  Queue : Queue_Type;
  Item   : Positive;

begin
  Enqueue (Item => 1, Queue => Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue); -- will produce a "Queue underflow"

exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow  => Put ("Queue overflow");
end Queue_Test_Generic;
```

Instantiate generic package



# *Introduction & Languages*

## *A generic queue test program*

```
with Queue_Pack_Generic; -- cannot apply 'use' clause here
with Ada.Text_IO          ; use Ada.Text_IO;
procedure Queue_Test_Generic is
  package Queue_Pack_Positive is
    new Queue_Pack_Generic (Element => Positive);
  use Queue_Pack_Positive; -- 'use' clause can be applied to instantiated package
  Queue : Queue_Type;
  Item   : Positive;
begin
  Enqueue (Item => 1, Queue => Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue); -- will produce a "Queue underflow"
exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow  => Put ("Queue overflow");
end Queue_Test_Generic;
```

... anything on this slide  
still not perfectly clear?



# *Introduction & Languages*

## *A generic queue specification*

**generic**

```
  type Element is private;
  package Queue_Pack_Generic is
    QueueSize: constant Integer := 10;
    type Queue_Type is limited private;
    procedure Enqueue (Item: Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
    function Is_Empty (Queue : Queue_Type) return Boolean;
    function Is_Full (Queue : Queue_Type) return Boolean;
    Queueoverflow, Queueunderflow : exception;
  private
    type Marker is mod QueueSize;
    type List is array (Marker) of Element;
    type Queue_Type is record
      Top, Free : Marker := Marker'First;
      Is_Empty : Boolean := True;
      Elements : List;
    end record;
  end Queue_Pack_Generic;
```

None of the packages so far can be used in a concurrent environment.



# *Introduction & Languages*

*Ada*

## *Access routines for concurrent systems*

... introducing:

- **Protected objects**
- **Entry guards**
- **Side-effecting (mutually exclusive) entry and procedure calls**
- **Side-effect-free (concurrent) function calls**

# A generic protected queue specification

```
generic
    type Element is private;
    type Index   is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Protected_Generic is
    type Queue_Type is limited private;
    protected type Protected_Queue is
        entry Enqueue (Item : Element);
        entry Dequeue (Item : out Element);
        procedure Empty_Queue;
        function Is_Empty return Boolean;
        function Is_Full  return Boolean;
    private
        Queue : Queue_Type;
    end Protected_Queue;
private
    type List is array (Index) of Element;
    type Queue_Type is record
        Top, Free : Index := Index'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;
end Queue_Pack_Protected_Generic;
```

# A generic protected queue specification

```
generic
    type Element is private;
    type Index   is mod <>; -- Modulo defines size of the queue.

package Queue_Pack_Protected_Generic is
    type Queue_Type is limited private;
    protected type Protected_Queue is
        entry Enqueue (Item : Element);
        entry Dequeue (Item : out Element);
        procedure Empty_Queue;
        function Is_Empty return Boolean;
        function Is_Full  return Boolean;
    private
        Queue : Queue_Type;
    end Protected_Queue;

private
    type List is array (Index) of Element;
    type Queue_Type is record
        Top, Free : Index := Index'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;
end Queue_Pack_Protected_Generic;
```



Generic components of the package:  
Element can be anything  
while the Index need to  
be a modulo type.

# A generic protected queue specification

generic

```
  type Element is private;
  type Index   is mod <>; -- Modulo defines size of the queue.
```

package Queue\_Pack\_Protected\_Generic is

```
  type Queue_Type is limited private;
```

```
  protected type Protected_Queue is
```

```
    entry Enqueue (Item : Element);
    entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full   return Boolean;
```

```
  private
```

```
    Queue : Queue_Type;
```

```
  end Protected_Queue;
```

private

```
  type List is array (Index) of Element;
```

```
  type Queue_Type is record
```

```
    Top, Free : Index   := Index'First;
    Is_Empty : Boolean := True;
    Elements : List;
```

```
  end record;
```

```
end Queue_Pack_Protected_Generic;
```

Queue is protected for safe concurrent access.

Three categories of access routines are distinguished by the keywords:  
entry, procedure, function

# A generic protected queue specification

```
generic
    type Element is private;
    type Index   is mod <>; -- Modulo defines size of the queue.

package Queue_Pack_Protected_Generic is

    type Queue_Type is limited private;

    protected type Protected_Queue is
        entry Enqueue (Item : Element);
        entry Dequeue (Item : out Element);
        procedure Empty_Queue;
        function Is_Empty return Boolean;
        function Is_Full   return Boolean;
    private
        Queue : Queue_Type;
    end Protected_Queue;

private
    type List is array (Index) of Element;
    type Queue_Type is record
        Top, Free : Index := Index'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;
end Queue_Pack_Protected_Generic;
```

Procedures are **mutually exclusive** to all other access routines.

Rationale:

Procedures can modify the protected data.

Hence they need a guarantee for exclusive access.

# A generic protected queue specification

```
generic
    type Element is private;
    type Index   is mod <>; -- Modulo defines size of the queue.

package Queue_Pack_Protected_Generic is

    type Queue_Type is limited private;

    protected type Protected_Queue is
        entry Enqueue (Item : Element);
        entry Dequeue (Item : out Element);
        procedure Empty_Queue;
        function Is_Empty return Boolean;
        function Is_Full   return Boolean;

    private
        Queue : Queue_Type;
    end Protected_Queue;

private
    type List is array (Index) of Element;
    type Queue_Type is record
        Top, Free : Index := Index'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;
end Queue_Pack_Protected_Generic;
```

Functions are **mutually exclusive** to procedures and entries, yet **concurrent** to other functions.

Rationale:

The compiler enforces those functions to be side-effect-free with respect to the protected data.

Hence concurrent access can be granted among functions without risk.

# A generic protected queue specification

```
generic
    type Element is private;
    type Index   is mod <>; -- Modulo defines size of the queue.

package Queue_Pack_Protected_Generic is

    type Queue_Type is limited private;

    protected type Protected_Queue is
        entry Enqueue (Item : Element);
        entry Dequeue (Item : out Element);
        procedure Empty_Queue;
        function Is_Empty return Boolean;
        function Is_Full  return Boolean;
    private
        Queue : Queue_Type;
    end Protected_Queue;

    private
        type List is array (Index) of Element;
        type Queue_Type is record
            Top, Free : Index := Index'First;
            Is_Empty : Boolean := True;
            Elements : List;
        end record;
    end Queue_Pack_Protected_Generic;
```

Entries are **mutually exclusive** to all other access routines and also provide one **guard** per entry which need to evaluate to True before entry is granted.

The **guard expressions** are defined in the implementation part.

Rationale:

Entries can be blocking even if the protected object itself is unlocked.

Hence a separate task waiting queue is provided per entry.

# A generic protected queue specification

```
generic
    type Element is private;
    type Index   is mod <>; -- Modulo defines size of the queue.

package Queue_Pack_Protected_Generic is

    type Queue_Type is limited private;

    protected type Protected_Queue is
        entry Enqueue (Item : Element);
        entry Dequeue (Item : out Element);
        procedure Empty_Queue;
        function Is_Empty return Boolean;
        function Is_Full  return Boolean;
    private
        Queue : Queue_Type;
    end Protected_Queue;

private
    type List is array (Index) of Element;
    type Queue_Type is record
        Top, Free : Index := Index'First;
        Is_Empty : Boolean := True;
        Elements : List;
    end record;
end Queue_Pack_Protected_Generic;
```

... anything on this slide  
still not perfectly clear?

# *A generic protected queue implementation*

```
package body Queue_Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
      begin
        Queue.Elements (Queue.Free) := Item; Queue.Free := Index'Succ (Queue.Free);
        Queue.Is_Empty := False;
      end Enqueue;

    entry Dequeue (Item : out Element) when not Is_Empty is
      begin
        Item := Queue.Elements (Queue.Top); Queue.Top := Index'Succ (Queue.Top);
        Queue.Is_Empty := Queue.Top = Queue.Free;
      end Dequeue;

    procedure Empty_Queue is
      begin
        Queue.Top := Index'First; Queue.Free := Index'First; Queue.Is_Empty := True;
      end Empty_Queue;

    function Is_Empty return Boolean is (Queue.Is_Empty);
    function Is_Full   return Boolean is
      (not Queue.Is_Empty and then Queue.Top = Queue.Free);
    end Protected_Queue;
  end Queue_Pack_Protected_Generic;
```

# A generic protected queue *implementation*

```
package body Queue_Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
      begin
        Queue.Elements (Queue.Free) := Item; Queue.Free := Index'Succ (Queue.Free);
        Queue.Is_Empty := False;
      end Enqueue;

    entry Dequeue (Item : out Element) when not Is_Empty is
      begin
        Item := Queue.Elements (Queue.Top); Queue.Top := Index'Succ (Queue.Top);
        Queue.Is_Empty := Queue.Top = Queue.Free;
      end Dequeue;
    procedure Empty_Queue is
      begin
        Queue.Top := Index'First; Queue.Free := Index'First; Queue.Is_Empty := True;
      end Empty_Queue;
    function Is_Empty return Boolean is
      begin
        return Queue.Top <= Queue.Free;
      end Is_Empty;
    function Is_Full return Boolean is
      begin
        return Queue.Top = Queue.Free;
      end Is_Full;
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  end Protected_Queue;
end Queue_Pack_Protected_Generic;
```

**Guard expressions**  
follow after when in the implementation of entries.

Tasks are automatically blocked or released depending on the state of the guard.  
Guard expressions are re-evaluated on exiting an entry or procedure  
(no point to re-check them at any other time).  
Exactly one waiting task on one entry is released.

# *A generic protected queue implementation*

```
package body Queue_Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
      begin
        Queue.Elements (Queue.Free) := Item; Queue.Free := Index'_succ (Queue.Free);
        Queue.Is_Empty := False;
      end Enqueue;

    entry Dequeue (Item : out Element) when not Is_Empty is
      begin
        Item := Queue.Elements (Queue.Top); Queue.Top := Index'_succ (Queue.Top);
        Queue.Is_Empty := Queue.Top = Queue.Free;
      end Dequeue;

    procedure Empty_Queue is
      begin
        Queue.Top := Index'First; Queue.Free := Index'First; Queue.Is_Empty := True;
      end Empty_Queue;

    function Is_Empty return Boolean is (Queue.Is_Empty);
    function Is_Full  return Boolean is
      (not Queue.Is_Empty and then Queue.Top = Queue.Free);
    end Protected_Queue;
  end Queue_Pack_Protected_Generic;
```

... anything on this slide  
still not perfectly clear?

# *A generic protected queue test program*

```
with Ada.Task_Identification;      use Ada.Task_Identification;
with Ada.Text_IO;                  use Ada.Text_IO;
with Queue_Pack_Protected_Generic;
procedure Queue_Test_Protected_Generic is
    type Queue_Size is mod 3;
    package Queue_Pack_Protected_Character is
        new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
    use Queue_Pack_Protected_Character;
    Queue : Protected_Queue;
    type Task_Index is range 1 .. 3;
    task type Producer;
    task type Consumer;
    Producers : array (Task_Index) of Producer;
    Consumers : array (Task_Index) of Consumer;
    (...)

begin
    null;
end Queue_Test_Protected_Generic;
```

# A generic protected queue test program

```
with Ada.Task_Identification;      use Ada.Task_Identification;
with Ada.Text_IO;                  use Ada.Text_IO;
with Queue_Pack_Protected_Generic;
procedure Queue_Test_Protected_Generic is
    type Queue_Size is mod 3;
    package Queue_Pack_Protected_Character is
        new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
    use Queue_Pack_Protected_Character;
    Queue : Protected_Queue;
    type Task_Index is range 1 .. 3;
    task type Producer;
    task type Consumer;
    Producers : array (Task_Index) of Producer;
    Consumers : array (Task_Index) of Consumer;
    (...)

begin
    null;
end Queue_Test_Protected_Generic;
```

If more than one instance of a specific task is to be run then a **task type** (as opposed to a concrete task) is declared.

# A generic protected queue test program

```
with Ada.Task_Identification;      use Ada.Task_Identification;
with Ada.Text_IO;                  use Ada.Text_IO;
with Queue_Pack_Protected_Generic;
procedure Queue_Test_Protected_Generic is
    type Queue_Size is mod 3;
    package Queue_Pack_Protected_Character is
        new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
    use Queue_Pack_Protected_Character;
    Queue : Protected_Queue;
    type Task_Index is range 1 .. 3;
    task type Producer;
    task type Consumer;
    Producers : array (Task_Index) of Producer;
    Consumers : array (Task_Index) of Consumer;
    (...)

begin
    null;
end Queue_Test_Protected_Generic;
```

Multiple instances of a task can be instantiated e.g. by declaring an array of this task type.

Tasks are started right when such an array is created.

# A generic protected queue test program

```
with Ada.Task_Identification;      use Ada.Task_Identification;
with Ada.Text_IO;                  use Ada.Text_IO;
with Queue_Pack_Protected_Generic;
procedure Queue_Test_Protected_Generic is
    type Queue_Size is mod 3;
    package Queue_Pack_Protected_Character is
        new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
    use Queue_Pack_Protected_Character;
    Queue : Protected_Queue;
    type Task_Index is range 1 .. 3;
    task type Producer;
    task type Consumer;
    Producers : array (Task_Index) of Producer; ←
    Consumers : array (Task_Index) of Consumer;
    (...)

begin
    null;
end Queue_Test_Protected_Generic;
```

These declarations spawned off all the production code.

Often there are no statements for the “main task”  
(here explicitly stated by a null statement).

This task is prevented from terminating though until all tasks inside its scope terminated.

# *A generic protected queue test program*

```
with Ada.Task_Identification;      use Ada.Task_Identification;
with Ada.Text_IO;                  use Ada.Text_IO;
with Queue_Pack_Protected_Generic;
procedure Queue_Test_Protected_Generic is
    type Queue_Size is mod 3;
    package Queue_Pack_Protected_Character is
        new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
    use Queue_Pack_Protected_Character;
    Queue : Protected_Queue;
    type Task_Index is range 1 .. 3;
    task type Producer;
    task type Consumer;
    Producers : array (Task_Index) of Producer;
    Consumers : array (Task_Index) of Consumer;
    (...)

begin
    null;
end Queue_Test_Protected_Generic;
```

... anything on this slide  
still not perfectly clear?

## A generic protected queue test program (cont.)

```
subtype Some_Characters is Character range 'a' .. 'f';

task body Producer is
begin
  for Ch in Some_Characters loop
    Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " &
              (if Queue.Is_Empty then "EMPTY" else "not empty") &
              " and " &
              (if Queue.Is_Full then "FULL" else "not full") &
              " and prepares to add: " & Character'Image (Ch) &
              " to the queue.");
    Queue.Enqueue (Ch); -- task might be blocked here!
  end loop;
  Put_Line ("----- Task " & Image (Current_Task) & " terminates.");
end Producer;
```

## A generic protected queue test program (cont.)

```
subtype Some_Characters is Character range 'a' .. 'f';
```

**task body** Producer **is**

**begin**

for Ch **in** Some\_Characters **loop**

Put\_Line ("Task " & Image (Current\_Task) & " finds the queue to be " &

(**if** Queue.Is\_Empty **then** "EMPTY" **else** "not empty") &  
" and " &

(**if** Queue.Is\_Full **then** "FULL" **else** "not full") &  
" and prepares to add: " & Character'Image (Ch) &  
" to the queue.");

Queue.Enqueue (Ch); -- task might be blocked here!

**end loop;**

Put\_Line ("----- Task " & Image (Current\_Task) & " terminates.");

**end Producer;**

The executable code for a task is provided in its body.

## A generic protected queue test program (cont.)

```
subtype Some_Characters is Character range 'a' .. 'f';

task body Producer is
begin
  for Ch in Some_Characters loop
    Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " &
              (if Queue.Is_Empty then "EMPTY" else "not empty") &
              " and " &
              (if Queue.Is_Full then "FULL" else "not full") &
              " and prepares to add: " & Character'Image (Ch) &
              " to the queue.");
    Queue.Enqueue (Ch); -- task might be blocked here!
  end loop;
  Put_Line ("----- Task " & Image (Current_Task) & " terminates.");
end Producer;
```

There are three of those tasks  
and they are all 'hammering'  
the queue at full CPU speed.

## A generic protected queue test program (cont.)

```
subtype Some_Characters is Character range 'a' .. 'f';

task body Producer is
begin
  for Ch in Some_Characters loop
    Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " &
              (if Queue.Is_Empty then "EMPTY" else "not empty") &
              " and " &
              (if Queue.Is_Full then "FULL" else "not full") &
              " and prepares to add: " & Character'Image (Ch) &
              " to the queue.");
    Queue.Enqueue (Ch); -- task might be blocked here!
  end loop;
  Put_Line ("----- Task " & Image (Current_Task) & " terminates.");
end Producer;
```

Tasks automatically terminate once they reach their end declaration  
(and all inner tasks are terminated).

## A generic protected queue test program (cont.)

```
subtype Some_Characters is Character range 'a' .. 'f';

task body Producer is
begin
  for Ch in Some_Characters loop
    Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " &
              (if Queue.Is_Empty then "EMPTY" else "not empty") &
              " and " &
              (if Queue.Is_Full then "FULL" else "not full") &
              " and prepares to add: " & Character'Image (Ch) &
              " to the queue.");
    Queue.Enqueue (Ch); -- task might be blocked here!
  end loop;
  Put_Line ("----- Task " & Image (Current_Task) & " terminates.");
end Producer;
```

... anything on this slide  
still not perfectly clear?

## A generic protected queue test program (cont.)

```
task body Consumer is
    Item      : Character;
    Counter   : Natural := 0;
begin
    loop
        Queue.Dequeue (Item); -- task might be blocked here!
        Counter := Natural'Succ (Counter);
        Put_Line ("Task " & Image (Current_Task) &
                  " received: " & Character'Image (Item) &
                  " and the queue appears to be " &
                  (if Queue.Is_Empty then "EMPTY" else "not empty") &
                  " and " &
                  (if Queue.Is_Full  then "FULL" else "not full") &
                  " afterwards.");
        exit when Item = Some_Characters'Last;
    end loop;
    Put_Line ("----- Task " & Image (Current_Task) &
              " terminates and received" & Natural'Image (Counter) & " items.");
end Consumer;
```

## A generic protected queue test program (cont.)

```
task body Consumer is
    Item      : Character;
    Counter   : Natural := 0;
begin
    loop
        Queue.Dequeue (Item); -- task might be blocked here!
        Counter := Natural'Succ (Counter);
        Put_Line ("Task " & Image (Current_Task) &
                  " received: " & Character'Image (Item) &
                  " and the queue appears to be " &
                  (if Queue.Is_Empty then "EMPTY" else "not empty") &
                  " and " &
                  (if Queue.Is_Full  then "FULL" else "not full") &
                  " afterwards.");
        exit when Item = Some_Characters'Last;
    end loop;
    Put_Line ("----- Task " & Image (Current_Task) &
              " terminates and received" & Natural'Image (Counter) & " items.");
end Consumer;
```

Another three tasks and are all  
'hammering' the queue at this  
end and at full CPU speed.

## A generic protected queue test program (cont.)

```
task body Consumer is
    Item      : Character;
    Counter   : Natural := 0;
begin
    loop
        Queue.Dequeue (Item); -- task might be blocked here!
        Counter := Natural'Succ (Counter);
        Put_Line ("Task " & Image (Current_Task) &
                  " received: " & Character'Image (Item) &
                  " and the queue appears to be " &
                  (if Queue.Is_Empty then "EMPTY" else "not empty") &
                  " and " &
                  (if Queue.Is_Full  then "FULL" else "not full") &
                  " afterwards.");
        exit when Item = Some_Characters'Last;
    end loop;
    Put_Line ("----- Task " & Image (Current_Task) &
              " terminates and received" & Natural'Image (Counter) & " items.");
end Consumer;
```

... anything on this slide  
still not perfectly clear?

# **A generic protected queue test program (output)**

Task producers(1) finds the queue to be EMPTY and not full and prepares to add: ‘a’ to the queue.  
Task producers(1) finds the queue to be not empty and not full and prepares to add: ‘b’ to the queue.  
Task producers(1) finds the queue to be not empty and not full and prepares to add: ‘c’ to the queue.  
Task producers(1) finds the queue to be not empty and FULL and prepares to add: ‘d’ to the queue.  
Task producers(2) finds the queue to be not empty and FULL and prepares to add: ‘a’ to the queue.  
Task producers(3) finds the queue to be not empty and FULL and prepares to add: ‘a’ to the queue.  
Task consumers(1) received: ‘a’ and the queue appears to be not empty and FULL afterwards.  
Task consumers(1) received: ‘b’ and the queue appears to be not empty and FULL afterwards.  
Task consumers(1) received: ‘c’ and the queue appears to be not empty and FULL afterwards.  
Task consumers(1) received: ‘d’ and the queue appears to be not empty and not full afterwards.  
Task consumers(1) received: ‘a’ and the queue appears to be not empty and not full afterwards.  
..  
<---- Task producers(1) terminates.  
..  
Task consumers(3) received: ‘b’ and the queue appears to be EMPTY and not full afterwards.  
<---- Task consumers(2) terminates and received 1 items.  
..  
<---- Task producers(2) terminates.  
..  
<---- Task producers(3) terminates.  
..  
<---- Task consumers(1) terminates and received 12 items.  
<---- Task consumers(3) terminates and received 5 items.

What is going on here?

# **A generic protected queue test program (another output)**

Task producers(1) finds the queue to be EMPTY and not full and prepares to add: ‘a’ to the queue.  
Task producers(2) finds the queue to be EMPTY and not full and prepares to add: ‘a’ to the queue.  
Task producers(1) finds the queue to be not empty and not full and prepares to add: ‘b’ to the queue.  
Task consumers(1) received: ‘a’ and the queue appears to be EMPTY and not full afterwards.  
Task producers(3) finds the queue to be EMPTY and not full and prepares to add: ‘a’ to the queue.  
Task producers(1) finds the queue to be EMPTY and not full and prepares to add: ‘c’ to the queue.  
Task producers(2) finds the queue to be EMPTY and not full and prepares to add: ‘b’ to the queue.  
Task consumers(2) received: ‘a’ and the queue appears to be EMPTY and not full afterwards.  
Task consumers(3) received: ‘b’ and the queue appears to be EMPTY and not full afterwards.  
..  
----- Task producers(1) terminates.  
Task producers(2) finds the queue to be not empty and FULL and prepares to add: ‘f’ to the queue.  
Task consumers(2) received: ‘f’ and the queue appears to be not empty and not full afterwards.  
Task consumers(3) received: ‘e’ and the queue appears to be EMPTY and not full afterwards.  
Task producers(3) finds the queue to be not empty and not full and prepares to add: ‘f’ to the queue.  
Task consumers(1) received: ‘d’ and the queue appears to be not empty and not full afterwards.  
----- Task producers(2) terminates.  
----- Task consumers(2) terminates and received 5 items.  
Task consumers(3) received: ‘e’ and the queue appears to be not empty and not full afterwards.  
----- Task producers(3) terminates.  
Task consumers(1) received: ‘f’ and the queue appears to be not empty and not full afterwards.  
Task consumers(3) received: ‘f’ and the queue appears to be EMPTY and not full afterwards.  
----- Task consumers(1) terminates and received 6 items.  
----- Task consumers(3) terminates and received 7 items.

Does this make any sense?

# *A protected, generic queue specification*

```
generic
  type Element is private;
  Queue_Size : Positive := 10;
package Queue_Pack_Protected_Generic is
  type Queue_Type is limited private;
  protected type Protected_Queue is
    entry Enqueue (Item : Element);
    entry Dequeue (Item : out Element);
    function Is_Empty return Boolean;
    function Is_Full return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;

private
  subtype Marker is Natural range 0 .. Queue_Size - 1;
  type List is array (Marker'Range) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;
```

All processing is done in/by the client tasks.  
☞ Can the processing of queue internals  
alternatively be done by a dedicated task?



# *Introduction & Languages*

*Ada*

## *Service tasks (Message passing)*

... introducing:

- **Select statements.**
- **Rendezvous** (synchronous message passing).
- **Automatic termination.**



# *Introduction & Languages*

## *An queue task specification*

```
generic
  type Element is private;
  Queue_Size : Positive := 10;
package Queue_Pack_Task_Generic is
  task type Queue_Task is
    entry Enqueue (Item : Element);
    entry Dequeue (Item : out Element);
    entry Is_Empty (Result : out Boolean);
    entry Is_Full   (Result : out Boolean);
  end Queue_Task;
end Queue_Pack_Task_Generic;
```



# *Introduction & Languages*

## *An queue task specification*

```
generic
  type Element is private;
  Queue_Size : Positive := 10;
package Queue_Pack_Task_Generic is
  task type Queue_Task is
    entry Enqueue (Item : Element);
    entry Dequeue (Item : out Element);
    entry Is_Empty (Result : out Boolean);
    entry Is_Full   (Result : out Boolean);
  end Queue_Task;
end Queue_Pack_Task_Generic;
```

A type for an active task servicing a queue.

All communication via synchronous message passing  
between the client tasks and the service task.



# *Introduction & Languages*

## *An queue task **implementation***

```
package body Queue_Pack_Task_Generic is
  task body Queue_Task is
    subtype Marker is Natural range 0 .. Queue_Size - 1;
    type List is array (Marker'Range) of Element;
    type Queue_Type is record
      Top, Free : Marker := Marker'First;
      Is_Empty : Boolean := True;
      Elements : List;
    end record;
    Queue : Queue_Type;
    function Is_Empty return Boolean is
      (Queue.Is_Empty);
    function Is_Full return Boolean is
      (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  (...)
```

Data structures and functions  
are declared local to the task.



# *Introduction & Languages*

## *An queue task implementation (cont.)*

(...)

**begin**

**loop**

**select**

**when not** Is\_Full =>

**accept** Enqueue (Item : Element) **do**

        Queue.Elements (Queue.Free) := Item;

**end** Enqueue;

    Queue.Free := (Queue.Free + 1) **mod** Queue\_Size;

    Queue.IsEmpty := False;

**or**

**when not** Is\_Empty =>

**accept** Dequeue (Item : **out** Element) **do**

        Item := Queue.Elements (Queue.Top);

**end** Dequeue;

    Queue.Top := (Queue.Top + 1) **mod** Queue\_Size;

    Queue.IsEmpty := Queue.Top = Queue.Free;

(...)



# Introduction & Languages

## An queue task implementation (cont.)

(...)

begin

loop

select

when not Is\_Full =>

accept Enqueue (Item : Element) do

Queue.Elements (Queue.Free) := Item;

end Enqueue;

Queue.Free := (Queue.Free + 1) mod Queue\_Size;

Queue.IsEmpty := False;

or

when not Is\_Empty =>

accept Dequeue (Item : out Element) do

Item := Queue.Elements (Queue.Top);

end Dequeue;

Queue.Top := (Queue.Top + 1) mod Queue\_Size;

Queue.IsEmpty := Queue.Top = Queue.Free;

(...)

If guards are “closed” tasks will be queued (suspended).

Operations are only accepted once pre-conditions are fulfilled.



# *Introduction & Languages*

## *An queue task implementation (cont.)*

(...)

```
begin
  loop
    select
      when not Is_Full =>
        accept Enqueue (Item : Element) do
          Queue.Elements (Queue.Free) := Item;
          end Enqueue;
          Queue.Free := (Queue.Free + 1) mod Queue_Size;
          Queue.IsEmpty := False;
      or
      when not Is_Empty =>
        accept Dequeue (Item : out Element) do
          Item := Queue.Elements (Queue.Top);
          end Dequeue;
          Queue.Top := (Queue.Top + 1) mod Queue_Size;
          Queue.IsEmpty := Queue.Top = Queue.Free;
    end select;
  end loop;
end;
```

Tasks are synchronized inside the rendezvous blocks.

(...)



# Introduction & Languages

## An queue task implementation (cont.)

(...)

**begin**

**loop**

**select**

**when not Is\_Full =>**

**accept Enqueue (Item : Element) do**

    Queue.Elements (Queue.Free) := Item;

**end Enqueue;**

    Queue.Free := (Queue.Free + 1) **mod Queue\_Size;**

    Queue.IsEmpty := False;

Client tasks are released (and  
continue concurrent operations).

Service task completes the  
operation on its own.

**or**

**when not Is\_Empty =>**

**accept Dequeue (Item : out Element) do**

    Item := Queue.Elements (Queue.Top);

**end Dequeue;**

    Queue.Top := (Queue.Top + 1) **mod Queue\_Size;**

    Queue.IsEmpty := Queue.Top = Queue.Free;

(...)



# *Introduction & Languages*

## *An queue task implementation (cont.)*

(...)

or

```
accept Is_Empty (Result : out Boolean) do
    Result := Is_Empty;
end Is_Empty;
```

or

```
accept Is_Full (Result : out Boolean) do
    Result := Is_Full;
end Is_Full;
```

or

```
terminate;
```

```
end select;
```

```
end loop;
```

```
end Queue_Task;
```

```
end Queue_Pack_Task_Generic;
```

Service task terminates if all potentially calling tasks are terminated themselves.

# *A generic queue task test program*

```
with Ada.Text_IO; use Ada.Text_IO;
with Queue_Pack_Task_Generic;
procedure Queue_Test_Protected_Generic is
  package Queue_Pack_Task_Character is
    new Queue_Pack_Task_Generic (Element => Character, Queue_Size => 12);
use Queue_Pack_Task_Character;
  Queue : Protected_Queue;
  task Producer is end Producer;
  task Consumer is end Consumer;
```

(...)

Identical to the test program  
for protected objects.

## A generic queue task test program (cont.)

```
task body Producer is
    subtype Lower is Character range 'a' .. 'z';
begin
    for Ch in Lower loop
        Queue.Enqueue (Ch); -- task might be blocked here!
    end loop;
end Producer;

task body Consumer is
    Item : Element;
begin
    loop
        select
            Queue.Dequeue (Item); -- task might be blocked here!
            Put ("Received: "); Put (Item); Put_Line ("!");
        or delay 0.001;
            exit; -- main task loop
        end select;
    end loop;
end Consumer;

begin
    null;
end Queue_Test_Protected_Generic;
```

These two calls are 'hammering'  
the queue task concurrently  
and at full CPU speed.

Identical to the test program  
for protected objects.



# *Introduction & Languages*

## *An queue task specification*

```
generic
    type Element is private;
    Queue_Size : Positive := 10;
package Queue_Pack_Task_Generic is
    task type Queue_Task is
        entry Enqueue (Item : Element);
        entry Dequeue (Item : out Element);
        entry Is_Empty (Result : out Boolean);
        entry Is_Full   (Result : out Boolean);
    end Queue_Task;
end Queue_Pack_Task_Generic;
```

While this allows for a high degree of concurrency, it does not lend itself to distributed communication directly.



# *Introduction & Languages*

*Ada*

## *Abstract types & dispatching*

... introducing:

- **Abstract tagged types & subroutines (Interfaces)**
- Concrete implementation of abstract types
- **Dynamic dispatching** to different packages, tasks, protected types or partitions.
- **Synchronous message passing.**



# *Introduction & Languages*

*Ada*

## *Abstract types & dispatching*

... introducing:

- **Abstract tagged types & subroutines (Interfaces)**
- Concrete implementation of abstract types
- **Dynamic dispatching** to different packages, tasks, protected types or partitions.
- **Synchronous message passing.**

– Advanced topic –

**Proceed with caution!**



# *Introduction & Languages*

## *An abstract queue specification*

```
generic
  type Element is private;
package Queue_Pack_Abstract is
  type Queue_Interface is synchronized interface;
  procedure Enqueue (Q : in out Queue_Interface; Item : Element) is abstract;
  procedure Dequeue (Q : in out Queue_Interface; Item : out Element) is abstract;
end Queue_Pack_Abstract;
```



# *Introduction & Languages*

## *An abstract queue specification*

**generic**

```
  type Element is private;
package Queue_Pack_Abstract is
  type Queue_Interface is synchronized interface;
  procedure Enqueue (Q : in out Queue_Interface; Item : Element) is abstract;
  procedure Dequeue (Q : in out Queue_Interface; Item : out Element) is abstract;
end Queue_Pack_Abstract;
```

### *Motivation:*

Different, derived implementations  
(potentially on different computers)  
can be passed around and referred to with the  
same common interface as defined here.



# *Introduction & Languages*

## *An abstract queue specification*

generic

```
type Element is private;
package Queue_Pack_Abstract is
    type Queue_Interface is synchronized interface;
    procedure Enqueue (Q : in out Queue_Interface; Item : Element) is abstract;
    procedure Dequeue (Q : in out Queue_Interface; Item : out Element) is abstract;
end Queue_Pack_Abstract;
```

synchronized means that this interface can only be implemented by **synchronized entities** like **protected objects** (as seen above) or **synchronous message passing**.

**Abstract**, empty type definition which serves to define interface templates.



# *Introduction & Languages*

## *An abstract queue specification*

```
generic
  type Element is private;
package Queue_Pack_Abstract is
  type Queue_Interface is synchronized interface;
  procedure Enqueue (Q : in out Queue_Interface; Item : Element) is abstract;
  procedure Dequeue (Q : in out Queue_Interface; Item : out Element) is abstract;
end Queue_Pack_Abstract;
```

Abstract methods need to be overridden with concrete methods when a new type is derived from it.



# *Introduction & Languages*

## *An abstract queue specification*

```
generic
  type Element is private;
package Queue_Pack_Abstract is
  type Queue_Interface is synchronized interface;
  procedure Enqueue (Q : in out Queue_Interface; Item : Element) is abstract;
  procedure Dequeue (Q : in out Queue_Interface; Item : out Element) is abstract;
end Queue_Pack_Abstract;
```

... this does not require an implementation package (as all procedures are abstract)

... anything on this slide  
still not perfectly clear?



# *Introduction & Languages*

## *A concrete queue specification*

```
with Queue_Pack_Abstract;
generic
  with package Queue_Instance is new Queue_Pack_Abstract (<>);
  type Index is mod <>; -- Modulo defines size of the queue.

package Queue_Pack_Concrete is
  use Queue_Instance;
  type Queue_Type is limited private;
  protected type Protected_Queue is new Queue_Interface with
    overriding entry Enqueue (Item : Element);
    overriding entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full   return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  (...) -- as all previous private queue declarations
end Queue_Pack_Concrete;
```



# Introduction & Languages

## A concrete queue **specification**

```
with Queue_Pack_Abstract;
generic
  with package Queue_Instance is new Queue_Pack_Abstract (<>);
  type Index is mod <>; -- Modulo defines size of the queue.

package Queue_Pack_Concrete is
  use Queue_Instance;
  type Queue_Type is limited private;

  protected type Protected_Queue is new Queue_Interface with
    overriding entry Enqueue (Item : Element);
    overriding entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full   return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  (...) -- as all previous private queue declarations
end Queue_Pack_Concrete;
```

A generic package which takes another generic package as a parameter.



# Introduction & Languages

## A concrete queue **specification**

```
with Queue_Pack_Abstract;
generic
  with package Queue_Instance is new Queue_Pack_Abstract (<>);
  type Index is mod <>; -- Modulo defines size of the queue.

package Queue_Pack_Concrete is
  use Queue_Instance;
  type Queue_Type is limited private;

  protected type Protected_Queue is new Queue_Interface with
    overriding entry Enqueue (Item : Element);
    overriding entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full   return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  (...) -- as all previous private queue declarations
end Queue_Pack_Concrete;
```

A synchronous implementation of the abstract type Queue\_Interface

All abstract methods are **overridden** with concrete implementations.



# Introduction & Languages

## A *concrete queue specification*

```
with Queue_Pack_Abstract;
generic
  with package Queue_Instance is new Queue_Pack_Abstract (<>);
  type Index is mod <>; -- Modulo defines size of the queue.

package Queue_Pack_Concrete is
  use Queue_Instance;
  type Queue_Type is limited private;
  protected type Protected_Queue is new Queue_Interface with
    overriding entry Enqueue (Item : Element);
    overriding entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full   return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  (...) -- as all previous private queue declarations
end Queue_Pack_Concrete;
```

Other (non-overriding) methods can be added.



# Introduction & Languages

## A *concrete queue specification*

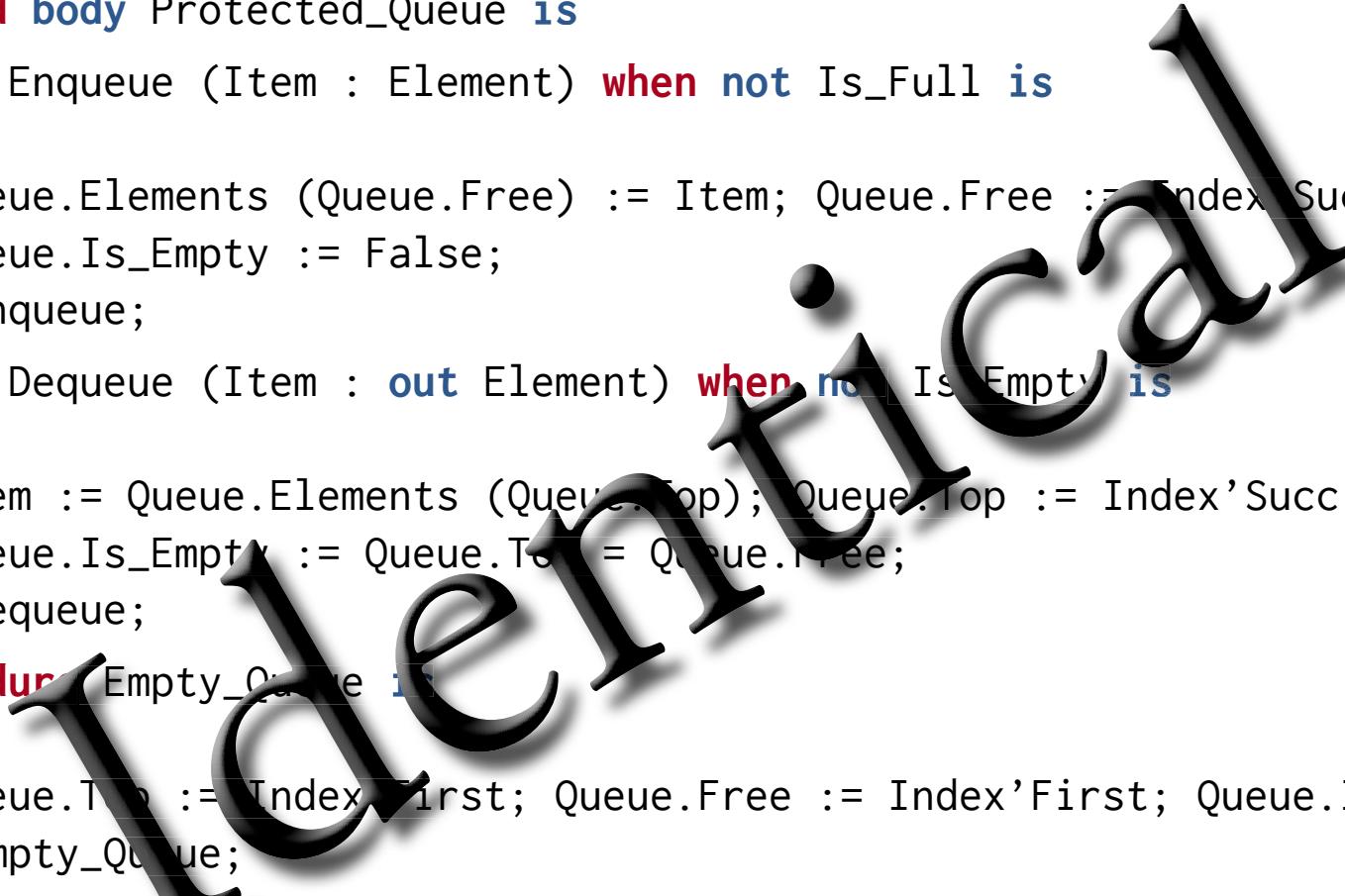
```
with Queue_Pack_Abstract;
generic
  with package Queue_Instance is new Queue_Pack_Abstract (<>);
  type Index is mod <>; -- Modulo defines size of the queue.

package Queue_Pack_Concrete is
  use Queue_Instance;
  type Queue_Type is limited private;
  protected type Protected_Queue is new Queue_Interface with
    overriding entry Enqueue (Item : Element);
    overriding entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full   return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  (...) -- as all previous private queue declarations
end Queue_Pack_Concrete;
```

... anything on this slide  
still not perfectly clear?

# A concrete queue *implementation*

```
package body Queue_Pack_Concrete is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
      begin
        Queue.Elements (Queue.Free) := Item; Queue.Free := Index'Succ (Queue.Free);
        Queue.Is_Empty := False;
      end Enqueue;
    entry Dequeue (Item : out Element) when not Is_Empty is
      begin
        Item := Queue.Elements (Queue.Top); Queue.Top := Index'Succ (Queue.Top);
        Queue.Is_Empty := Queue.Top = Queue.Free;
      end Dequeue;
    procedure Empty_Queue is
      begin
        Queue.Top := Index'First; Queue.Free := Index'First; Queue.Is_Empty := True;
      end Empty_Queue;
    function Is_Empty return Boolean is (Queue.Is_Empty);
    function Is_Full return Boolean is
      (not Queue.Is_Empty and then Queue.Top = Queue.Free);
    end Protected_Queue;
  end Queue_Pack_Concrete;
```



# A *dispatching test program*

```
with Ada.Text_IO;           use Ada.Text_IO;
with Queue_Pack_Abstract;
with Queue_Pack_Concrete;
procedure Queue_Test_Dispatching is
    package Queue_Pack_Abstract_Character is
        new Queue_Pack_Abstract (Character);
    use Queue_Pack_Abstract_Character;
    type Queue_Size is mod 3;
    package Queue_Pack_Character is
        new Queue_Pack_Concrete (Queue_Pack_Abstract_Character, Queue_Size);
    use Queue_Pack_Character;
    type Queue_Class is access all Queue_Interface'class;
    task Queue_Holder; -- could be on an individual partition / separate computer
    task Queue_User is -- could be on an individual partition / separate computer
        entry Send_Queue (Remote_Queue : Queue_Class);
    end Queue_User;
    (...)

begin
    null;
end Queue_Test_Dispatching;
```

# A *dispatching test program*

```
with Ada.Text_IO;           use Ada.Text_IO;
with Queue_Pack_Abstract;
with Queue_Pack_Concrete;
procedure Queue_Test_Dispatching is
    package Queue_Pack_Abstract_Character is
        new Queue_Pack_Abstract (Character);
    use Queue_Pack_Abstract_Character;
    type Queue_Size is mod 3;
    package Queue_Pack_Character is
        new Queue_Pack_Concrete (Queue_Pack_Abstract_Character, Queue_Size);
    use Queue_Pack_Character;
    type Queue_Class is access all Queue_Interface'class;
    task Queue_Holder; -- could be on an individual partition / separate computer
    task Queue_User is -- could be on an individual partition / separate computer
        entry Send_Queue (Remote_Queue : Queue_Class);
    end Queue_User;
    (...)

begin
    null;
end Queue_Test_Dispatching;
```

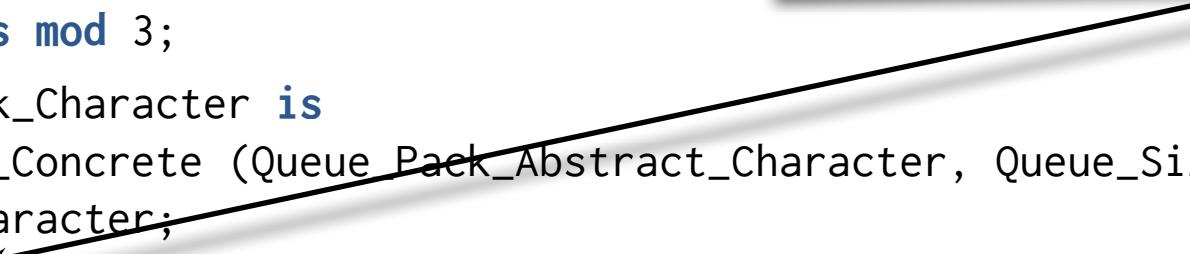
Sequence of instantiations

# A *dispatching test program*

```
with Ada.Text_IO;           use Ada.Text_IO;
with Queue_Pack_Abstract;
with Queue_Pack_Concrete;
procedure Queue_Test_Dispatching is
    package Queue_Pack_Abstract_Character is
        new Queue_Pack_Abstract (Character);
    use Queue_Pack_Abstract_Character;
    type Queue_Size is mod 3;
    package Queue_Pack_Character is
        new Queue_Pack_Concrete (Queue_Pack_Abstract_Character, Queue_Size);
    use Queue_Pack_Character;
    type Queue_Class is access all Queue_Interface'class;
    task Queue_Holder; -- could be on an individual partition / separate computer
    task Queue_User is -- could be on an individual partition / separate computer
        entry Send_Queue (Remote_Queue : Queue_Class);
    end Queue_User;
    (...)

begin
    null;
end Queue_Test_Dispatching;
```

Type which can refer to any instance of Queue\_Interface



# A *dispatching test program*

```
with Ada.Text_IO;           use Ada.Text_IO;
with Queue_Pack_Abstract;
with Queue_Pack_Concrete;
procedure Queue_Test_Dispatching is
    package Queue_Pack_Abstract_Character is
        new Queue_Pack_Abstract (Character);
    use Queue_Pack_Abstract_Character;
    type Queue_Size is mod 3;
    package Queue_Pack_Character is
        new Queue_Pack_Concrete (Queue_Pack_Abstract_Character, Queue_Size);
    use Queue_Pack_Character;
    type Queue_Class is access all Queue_Interface'class;
    task Queue_Holder; -- could be on an individual partition / separate computer
    task Queue_User is -- could be on an individual partition / separate computer
        entry Send_Queue (Remote_Queue : Queue_Class);
    end Queue_User;
    (...)
```

Declaring two concrete tasks.  
(Queue\_User has a synchronous message passing entry)

```
begin
    null;
end Queue_Test_Dispatching;
```

# A *dispatching test program*

```
with Ada.Text_IO;           use Ada.Text_IO;
with Queue_Pack_Abstract;
with Queue_Pack_Concrete;
procedure Queue_Test_Dispatching is
    package Queue_Pack_Abstract_Character is
        new Queue_Pack_Abstract (Character);
    use Queue_Pack_Abstract_Character;
    type Queue_Size is mod 3;
    package Queue_Pack_Character is
        new Queue_Pack_Concrete (Queue_Pack_Abstract_Character, Queue_Size);
    use Queue_Pack_Character;
    type Queue_Class is access all Queue_Interface'class;
    task Queue_Holder; -- could be on an individual partition / separate computer
    task Queue_User is -- could be on an individual partition / separate computer
        entry Send_Queue (Remote_Queue : Queue_Class);
    end Queue_User;
    (...)

begin
    null;
end Queue_Test_Dispatching;
```

... anything on this slide  
still not perfectly clear?

## A dispatching test program (cont.)

```
task body Queue_Holder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item         : Character;
begin
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;

task body Queue_User is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item         : Character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
    Local_Queue.all.Enqueue ('l');
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;
```

## A dispatching test program (cont.)

```
task body Queue_Holder is
```

```
    Local_Queue : constant Queue_Class := new Protected_Queue;  
    Item        : Character;
```

```
begin
```

```
    Queue_User.Send_Queue (Local_Queue);
```

```
    Local_Queue.all.Dequeue (Item);
```

Declaring local queues in each task.

```
    Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
```

```
end Queue_Holder;
```

```
task body Queue_User is
```

```
    Local_Queue : constant Queue_Class := new Protected_Queue;  
    Item        : Character;
```

```
begin
```

```
    accept Send_Queue (Remote_Queue : Queue_Class) do
```

```
        Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
```

```
        Local_Queue.all.Enqueue ('l');
```

```
    end Send_Queue;
```

```
    Local_Queue.all.Dequeue (Item);
```

```
    Put_Line ("Local dequeue (User) : " & Character'Image (Item));
```

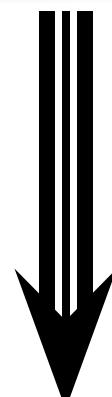
```
end Queue_User;
```

## A dispatching test program (cont.)

```
task body Queue_Holder is
    Local_Queue : constant Queue_Class := new Protected_Queue;
    Item         : Character;
begin
    Queue_User.Send_Queue (Local_Queue);
    Local_Queue.all.Dequeue (Item);
    Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;

task body Queue_User is
    Local_Queue : constant Queue_Class := new Protected_Queue;
    Item         : Character;
begin
    accept Send_Queue (Remote_Queue : Queue_Class) do
        Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
        Local_Queue.all.Enqueue ('l');
    end Send_Queue;
    Local_Queue.all.Dequeue (Item);
    Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;
```

Handing over the Holder's queue via synchronous message passing.



## A dispatching test program (cont.)

```
task body Queue_Holder is
    Local_Queue : constant Queue_Class := new Protected_Queue;
    Item         : Character;
begin
    Queue_User.Send_Queue (Local_Queue);
    Local_Queue.all.Dequeue (Item);
    Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;

task body Queue_User is
    Local_Queue : constant Queue_Class := new Protected_Queue;
    Item         : Character;
begin
    accept Send_Queue (Remote_Queue : Queue_Class) do
        Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
        Local_Queue.all.Enqueue ('l');
    end Send_Queue;
    Local_Queue.all.Dequeue (Item);
    Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;
```

Adding to both queues

## A dispatching test program (cont.)

Tasks could run on separate computers

```
task body Queue_Holder is
```

```
    Local_Queue : constant Queue_Class := new Protected_Queue;  
    Item         : Character;
```

```
begin
```

```
    Queue_User.Send_Queue (Local_Queue);
```

```
    Local_Queue.all.Dequeue (Item);
```

```
    Put_Line ("Local dequeue (Holder): " & Character'Image (Item));  
end Queue_Holder;
```

```
task body Queue_User is
```

```
    Local_Queue : constant Queue_Class := new Protected_Queue;  
    Item         : Character;
```

```
begin
```

```
    accept Send_Queue (Remote_Queue : Queue_Class) do
```

```
        Remote_Queue.all.Enqueue ('r'); - potentially a remote procedure call!
```

```
        Local_Queue.all.Enqueue ('l');
```

```
    end Send_Queue;
```

```
    Local_Queue.all.Dequeue (Item);
```

```
    Put_Line ("Local dequeue (User) : " & Character'Image (Item));
```

```
end Queue_User;
```

These two calls can be very different in nature:

The first call is potentially tunneled through a network to another computer and thus uses a **remote data structure**.

The second call is always a **local call** and using a **local data-structure**.

## A dispatching test program (cont.)

```
task body Queue_Holder is
    Local_Queue : constant Queue_Class := new Protected_Queue;
    Item         : Character;
begin
    Queue_User.Send_Queue (Local_Queue);
    Local_Queue.all.Dequeue (Item);
    Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;

task body Queue_User is
    Local_Queue : constant Queue_Class := new Protected_Queue;
    Item         : Character;
begin
    accept Send_Queue (Remote_Queue : Queue_Class) do
        Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
        Local_Queue.all.Enqueue ('l');
    end Send_Queue;
    Local_Queue.all.Dequeue (Item);
    Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;
```

Reading out 'r'

Reading out 'l'

## A dispatching test program (cont.)

```
task body Queue_Holder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item         : Character;
begin
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;

task body Queue_User is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item         : Character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
    Local_Queue.all.Enqueue ('l');
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;
```

... anything on this slide  
still not perfectly clear?



# *Introduction & Languages*

*Ada*

## *Coordinating concurrent reader tasks*

... introducing:

- **Entry families**
- **Entry attributes**

# *A protected, generic queues specification*

```
generic
  type Element is private;
  type Queue_Enum is (<>);
  Queue_Size : Positive := 10;
package Queues_Pack_Protected_Generic is
  type Queue_Type is limited private;
  protected type Protected_Queue is
    entry Enqueue          (Item : Element);
    entry Dequeue (Queue_Enum) (Item : out Element);
    function Is_Empty (Q : Queue_Enum) return Boolean;
    function Is_Full           return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
(...)
```

# *A protected, generic queues specification*

```
generic
  type Element is private;
  type Queue_Enum is (<>);
  Queue_Size : Positive := 10;

package Queues_Pack_Protected_Generic is

  type Queue_Type is limited private;

  protected type Protected_Queue is
    entry Enqueue          (Item : Element);
    entry Dequeue (Queue_Enum) (Item : out Element);
    function Is_Empty (Q : Queue_Enum) return Boolean;
    function Is_Full           return Boolean;

  private
    Queue : Queue_Type;

  end Protected_Queue;

(...)
```

No assumptions are made for a private type.

Any operations which are required on it besides copy and equality must be provided via this interface.

(<>) stands for any discrete type,  
i.e. all integer-derived types and enumerations.

# A protected, generic queues specification

```
generic
  type Element is private;
  type Queue_Enum is (<>);
  Queue_Size : Positive := 10;
package Queues_Pack_Protected_Generic is
  type Queue_Type is limited private;
  protected type Protected_Queue is
    entry Enqueue           (Item : Element);
    entry Dequeue (Queue_Enum) (Item : out Element);
    function Is_Empty (Q : Queue_Enum) return Boolean;
    function Is_Full          return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
(...)
```

This is actually a set of entries (“entry family”), one for every value in the type Queue\_Enum

Thus reading tasks can wait for their turn independently and will not be hindered by tasks waiting on other Dequeue entries.

## A protected, generic queues specification (cont.)

(...)

**private**

```
    subtype Marker is Natural range 0 .. Queue_Size - 1;
    type Markers is array (Queue_Enum) of Marker;
    type Readouts is array (Queue_Enum) of Boolean;
    All_Read : constant Readouts := (others => True);
    None_Read : constant Readouts := (others => False);

    type Element_and_Readouts is record
        Elem : Element; -- Initialized to invalids
        Reads : Readouts := All_Read;
    end record;
    type List is array (Marker'Range) of Element_and_Readouts;
    type Queue_Type is record
        Free : Marker := Marker'First;
        Readers : Markers := (others => Marker'First);
        Elements : List;
    end record;
end Queues_Pack_Protected_Generic;
```

Some data-structures to provide  
the impression of multiple queues  
which can be read independently.

# *A protected, generic queues implementation*

```
package body Queues_Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
      begin
        Queue.Elements (Queue.Free) := (Elem => Item, Reads => None_Read);
        Queue.Free := (Queue.Free + 1) mod Queue_Size;
    end Enqueue;

    entry Dequeue (for Q in Queue_Enum) (Item : out Element)
      when not Is_Empty (Q) and then (Enqueue'Count = 0 or else Is_Full) is
      begin
        Item := Queue.Elements (Queue.Readers (Q)).Elem;
        Queue.Elements (Queue.Readers (Q)).Reads (Q) := True;
        Queue.Readers (Q) := (Queue.Readers (Q) + 1) mod Queue_Size;
    end Dequeue;

    function Is_Empty (Q : Queue_Enum) return Boolean is
      (Queue.Elements (Queue.Readers (Q)).Reads (Q));
    function Is_Full return Boolean is
      (Queue.Elements (Queue.Free).Reads /= All_Read);
  end Protected_Queue;
end Queues_Pack_Protected_Generic;
```

# A protected, generic queues implementation

```
package body Queues_Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
      begin
        Queue.Elements (Queue.Free) := (Elem => Item, Reads => None_Read);
        Queue.Free := (Queue.Free + 1) mod Queue_Size;
      end Enqueue;

    entry Dequeue (for Q in Queue_Enum) (Item : out Element)
      when not Is_Empty (Q) and then (Enqueue'Count = 0 or else Is_Full) is
      begin
        Item := Queue.Elements (Queue.Readers (Q)).Elem;
        Queue.Elements (Queue.Readers (Q)).Reads (Q) := True;
        Queue.Readers (Q) := (Queue.Readers (Q) + 1) mod Queue_Size;
      end Dequeue;

    function Is_Empty (Q : Queue_Enum) return Boolean is
      (Queue.Elements (Queue.Readers (Q)).Reads (Q)) = 0;
    end Is_Empty;

    function Is_Full return Boolean is
      (Queue.Elements (Queue.Free).Reads / Queue.Elements (Queue.Free).Elem) = Queue_Elements;
    end Is_Full;
  end Protected_Queue;
end Queues_Pack_Protected_Generic;
```

The individual Dequeue entries open and close individually, depending on the fill status of the associated queue.

The also all give preference to the Enqueue entry.

# *A protected, generic queues implementation*

```
package body Queues_Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
      begin
        Queue.Elements (Queue.Free) := (Elem => Item, Reads => None_Read);
        Queue.Free := (Queue.Free + 1) mod Queue_Size;
      end Enqueue;
    entry Dequeue (for Q in Queue_Enum) (Item : out Element)
      when not Is_Empty (Q) and then (Enqueue'Count = 0 or else Is_Full) is
      begin
        Item := Queue.Elements (Queue.Readers (Q)).Elem;
        Queue.Elements (Queue.Readers (Q)).Reads (Q) := True;
        Queue.Readers (Q) := (Queue.Readers (Q) + 1) mod Queue_Size;
      end Dequeue;
    function Is_Empty (Q : Queue_Enum) return Boolean is
      (Queue.Elements (Queue.Readers (Q)).Reads (Q));
    function Is_Full return Boolean is
      (Queue.Elements (Queue.Free).Reads /= All_Read);
  end Protected_Queue;
end Queues_Pack_Protected_Generic;
```

The multi-reader data-structure makes full and empty detections easy.

# *A protected, generic queues test program*

```
with Queues_Pack_Protected_Generic;
with Ada.Text_IO; use Ada.Text_IO;
procedure Queues_Test_Protected_Generic is
    type Sequence      is (Ready, Set, Go);
    type Flight_States is (Take_Off, Cruising, Landing);
    package Queue_Pack_Protected_Character is
        new Queues_Pack_Protected_Generic
            (Element => Sequence, Queue_Enum => Flight_States, Queue_Size => 2);
    use Queue_Pack_Protected_Character;
    Queue : Protected_Queue;
    task type Avionics_Module is
        entry Provide_State (State : Flight_States);
    end Avionics_Module;
    Avionics : array (Flight_States) of Avionics_Module;
(...)
```

# *A protected, generic queues test program*

```
with Queues_Pack_Protected_Generic;
with Ada.Text_IO; use Ada.Text_IO;
procedure Queues_Test_Protected_Generic is
    type Sequence      is (Ready, Set, Go);
    type Flight_States is (Take_Off, Cruising, Landing);
    package Queue_Pack_Protected_Character is
        new Queues_Pack_Protected_Generic
            (Element => Sequence, Queue_Enum => Flight_States, Queue_Size => 2);
    use Queue_Pack_Protected_Character;
    Queue : Protected_Queue;
    task type Avionics_Module is
        entry Provide_State (State : Flight_States);
    end Avionics_Module;
    Avionics : array (Flight_States) of Avionics_Module;
    (...)
```

An array of tasks which we will use to read the individual queues.

## A protected, generic queues test program (cont.)

(...)

```
task body Avionics_Module is
    Local_State : Flight_States;
    Item         : Sequence;
begin
    accept Provide_State (State : Flight_States) do
        Local_State := State;
    end Provide_State;
    for Order in Sequence loop
        Queue.Dequeue (Local_State) (Item);
        Put_Line (Flight_States'Image (Local_State) &
                  " says:" & Sequence'Image (Item));
    end loop;
end Avionics_Module;

begin
    for State in Flight_States loop
        Avionics (State).Provide_State (State);
    end loop;
    for Order in Sequence loop
        Queue.Enqueue (Order);
        Put_Line ("Item added to queue: " & Sequence'Image (Order));
    end loop;
end Queues_Test_Protected_Generic;
```

## A protected, generic queues test program (cont.)

(...)

```
task body Avionics_Module is
    Local_State : Flight_States;
    Item         : Sequence;
begin
    accept Provide_State (State : Flight_States) do
        Local_State := State;
    end Provide_State;
    for Order in Sequence loop
        Queue.Dequeue (Local_State) (Item);
        Put_Line (Flight_States'Image (Local_State) &
                  " says:" & Sequence'Image (Item));
    end loop;
end Avionics_Module;

begin
    for State in Flight_States loop
        Avionics (State).Provide_State (State);
    end loop;
    for Order in Sequence loop
        Queue.Enqueue (Order);
        Put_Line ("Item added to queue: " & Sequence'Image (Order));
    end loop;
end Queues_Test_Protected_Generic;
```

Tasks are synchronized here (providing identities).

## A protected, generic queues test program (cont.)

(...)

```
task body Avionics_Module is
    Local_State : Flight_States;
    Item         : Sequence;
begin
    accept Provide_State (State : Flight_States) do
        Local_State := State;
    end Provide_State;
    for Order in Sequence loop
        Queue.Dequeue (Local_State) (Item);
        Put_Line (Flight_States'Image (Local_State) &
                  " says:" & Sequence'Image (Item));
    end loop;
end Avionics_Module;

begin
    for State in Flight_States loop
        Avionics (State).Provide_State (State);
    end loop;
    for Order in Sequence loop
        Queue.Enqueue (Order);
        Put_Line ("Item added to queue: " & Sequence'Image (Order));
    end loop;
end Queues_Test_Protected_Generic;
```

What will be the order  
of terminal outputs?

# *A protected, generic queues specification*

```
generic
  type Element is private;
  type Queue_Enum is (<>);
  Queue_Size : Positive := 10;
package Queues_Pack_Protected_Generic is
  type Queue_Type is limited private;
  protected type Protected_Queue is
    entry Enqueue          (Item : Element);
    entry Dequeue (Queue_Enum) (Item : out Element);
    function Is_Empty (Q : Queue_Enum) return Boolean;
    function Is_Full           return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
  (...)
```

For an actual real-time system, functional and temporal specifications (e.g. contracts) would be added and preferably proven.



# *Introduction & Languages*

## *Ada*

### *Ada language status*

- Established language standard with free and professionally supported compilers available for all major OSs and platforms.
- Emphasis on maintainability, high-integrity and efficiency.
- Stand-alone runtime environments for embedded systems.
- High integrity, real-time profiles part of the standard ↗ e.g. Ravenscar profile.



Boeing 787 cockpit (press release photo)

↗ Used in many large scale and/or high integrity projects

- Commonly used in aviation industry, high speed trains, metro-systems, space programs and military programs.
- ... also increasingly on small platforms / micro-controllers.



TGV, Renaud Chodkowski 2012



# *Introduction & Languages*

## *Real-Time Java*

*Specific Java engines and class libraries enhance:*

- **Threads:** Priorities, scheduling, and dispatching
- **Memory:** Controlled garbage collection and physical memory access
- **Synchronization:** Ordered queues, and priority ceiling protocols
- **Asynchronism:** Generalized asynchronous event handling, asynchronous transfer of control, timers, and an operational implementation of thread termination
  - ☞ All current real-time Java extensions keep the underlying, consequent object orientation. ☞ Predictability often questionable.
  - ☞ Some restrict the language standard, some extend it.



# *Introduction & Languages*

## *Real-Time Java*

### *Real-Time Specification for Java Versions 1.0.1 (2002) and 1.1 alpha 6 (2009)*

- Enhanced thread model (memory attributes, more precise specs).
- Enabling powerful and highly adaptive scheduling policies.
- Introducing scoped, immortal (keep the garbage collector out), and physical memory to Java (map to a physical architecture).
- Introducing timers, interrupts, and more exceptions.
- Higher resolution time model.
- Optional support for POSIX signals.

... despite being introduced in 2001, no sightings of industrial, hard real-time control systems implemented in RTS Java could be counted so far.  
(JamaicaVM might be the last implementation?)



# *Introduction & Languages*

## *Real-Time Java*

### *Real-Time Specification for Java*

- ☞ Standard library classes still rely on garbage collection!
  - ☞ i.e. usage of standard libraries destroys hard real-time integrity.
- ☞ RTSJ is backwards compatible
  - ☞ i.e. no syntactical extensions and no additional compiler checks (integers are still wrapping around, switch-statements need breaks etc.).
- ☞ Allows for different Java-engine implementations:
  - ☞ In terms of completeness: e.g. scheduling is not mandatory.
  - ☞ In terms of semantics: e.g. “instantiations per time span” can but does not need to imply equal distance intervals.
- ☞ Concept is still based on Java-style object oriented programming
  - ☞ Inheritance anomaly in concurrent systems needs to be considered carefully.



# *Introduction & Languages*

## *Esterel*

### ***Transformational* ↔ *Interactive* ↔ *Reactive***

- **Transformational** (functional) systems:

Generating outputs based on input and stop,  
utilizing no or only a small number of internal states.

- **Interactive** systems:

i.e. servers and other systems in long-term operation, requesting occasional inputs, and accepting service-calls, when there are resources to do so.

- **Reactive** (reflex) systems:

Systems which are reacting to external stimuli only (by generating other stimuli).  
Can be viewed as a predictable, functional system, which is listening to inputs continuously, while holding enough resources to ensure specified reaction times.



# *Introduction & Languages*

## *Esterel*

### *Control-oriented* $\leftrightarrow$ *Dataflow-oriented*

- **Dataflow-oriented:**

Continuous data-streams, functional processing (DSPs, filters, ...)  
☞ Typically high bandwidth

- **Control-oriented:**

Discrete signals controlling data-streams and processes  
☞ Typically low bandwidth



# *Introduction & Languages*

## *Esterel*

### *Esterel: Control-dominated Reactive Systems*

- **Real-Time process control:**
  - ☞ reaction to (sparse) stimuli in specified time-spans by emitting control signals.
- **Embedded systems / device control:**
  - ☞ local, discrete device control.
- **Complex systems control:**
  - ☞ supervision, moderation and control of complex data-streams.
- **Communication protocols:**
  - ☞ control/protocol part of communication systems.
- **Human-machine interface:**
  - ☞ switching modes, event and emergency handling.
- **Control logic (hardware):**
  - ☞ glue logic, interfaces, pipeline control, state machines.



# *Introduction & Languages*

## *Esterel*

### *Esterel: Strong synchrony or “zero delay” assumption*

In *logical* terms:

- ➡ All operations are **instantaneous!**



# *Introduction & Languages*

## *Esterel*

### *Esterel: Strong synchrony or “zero delay” assumption*

In *logical* terms:

- ☞ All operations are **instantaneous!**

In *physical* terms:

- ☞ There is **no observable delay** between stimuli and reaction!



# *Introduction & Languages*

## *Esterel*

### ***Esterel: Strong synchrony or “zero delay” assumption***

In *logical* terms:

- ☞ All operations are **instantaneous!**

In *physical* terms:

- ☞ There is **no observable delay** between stimuli and reaction!

In *computer science* terms:

- ☞ All operations are **finished before** the next input sampling.  
i.e. ☞ The base frequency is high enough such that the sampling can be considered instantaneous with respect to the physical system.

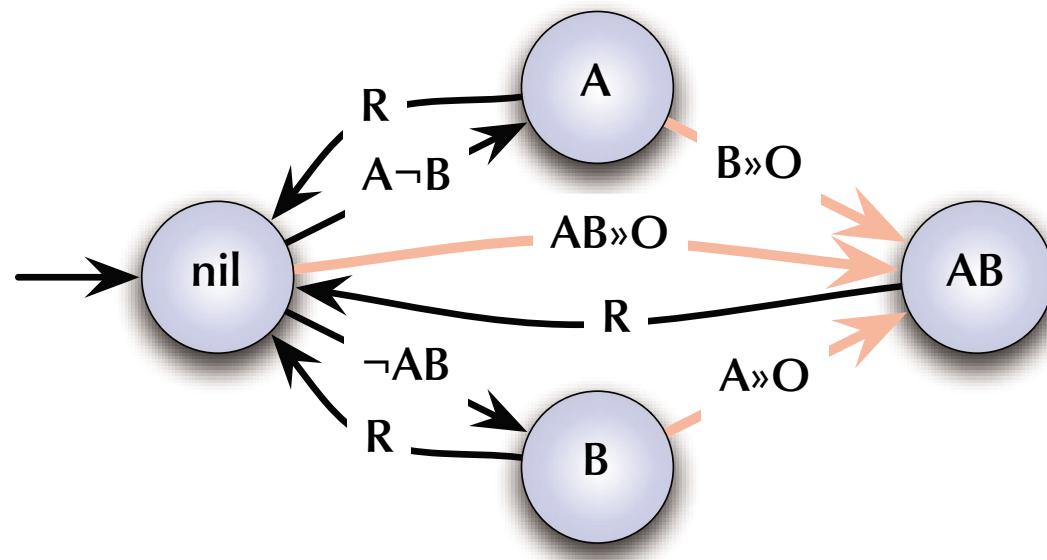


# Introduction & Languages

## Esterel

### *A simple, reactive, pure-signal example*

Mealy machine



Module in Esterel:

```
module A_and_B_gives_0;
  input A, B, R;
  output O;
  loop
    [await A || await B];
    emit O;
  each R;
end module;
```

## A simple, reactive integrator

Specification: a module should count the number of metres per second and emit this number as 'speed' once per second.

```
Module Speed;  
    input Metre, Second; relation Metre # Second;  
    output Speed: integer;  
  
    loop  
        var Distance := 0 : integer in  
  
            abort  
                every Metre do  
                    Distance := Distance + 1;  
                end every;  
                when Second do  
                    emit Speed (Distance);  
                end abort;  
            end var;  
        end loop;  
    end module;
```



-  These are exclusive
-  Hard aborted and restarted with every 'Metre' signal
-  Above block is hard aborted with every 'Second' signal



# *Introduction & Languages*

## *Esterel*

### *Immediate Reactions*

by default all synchronization points:

```
await <signal>;  
abort ... when <signal>;  
every <signal> do ... end every;  
loop ... each <signal>;
```

wait for the next signal occurrence ('*rising edge trigger*').

... yet with an additional 'immediate':

```
await immediate <signal>;  
abort ... when immediate <signal>;  
every immediate <signal> do ... end every;  
loop ... each immediate <signal>;
```

a currently active signal will trigger these statements ('*level trigger*').



# *Introduction & Languages*

## *Esterel*

### *Weak aborts*

by default a code block is aborted *immediately*, when <signal> occurs:

```
abort  
  [<statement>;]+  
  when <signal>;
```

Yet sometimes a finalization semantic

‘activate the code block **for** one last time, **when** <signal> occurs’  
is more useful and expressed in Esterel as:

```
weak abort  
  [<statement>;]+  
  when <signal>;
```

where the code block is now activated for a ‘final wish’, when <signal> occurs.

## A simple, reactive, *wrong* integrator

Specification: a module should count the number of metres per second and emit this number as ‘speed’ once per second.

```
module Speed;  
    input Metre, Second;  
    output Speed: integer;  
  
loop  
    var Distance := 0 : integer in  
  
        abort  
            every Metre do  
                Distance := Distance + 1;  
            end every;  
            when Second do  
                emit Speed (Distance);  
            end abort;  
        end var;  
    end loop;  
end module;
```



These are no longer exclusive



No guarantee that this block is active when ‘Metre’ occurs



Aborts immediately  
↳ even if a ‘Metre’ signal occurred simultaneously

## A simple, reactive, simultaneous signals integrator

'Metre' and 'Second' occur potentially simultaneously

☞ using '**weak abort**' to handle the simultaneous case :

```
module Speed;  
  input Metre, Second;  
  output Speed: integer;  
  
loop  
  var Distance := 0 : integer in  
  
    weak abort  
      every Metre do  
        Distance := Distance + 1;  
      end every;  
      when Second do  
        emit Speed (Distance);  
      end abort;  
    end var;  
  end loop;  
end module;
```



These are not exclusive



Block is always active  
when 'Metre' occurs



Activates block one last time with 'Second'  
☞ even if a 'Metre' signal occurred  
simultaneously, it will not be lost

## A simple, reactive, simultaneous signals integrator

'Metre' and 'Second' occur potentially simultaneously

☞ using '**every immediate**' to handle the simultaneous case :

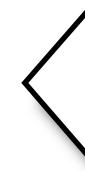
```
module Speed;  
  input Metre, Second;  
  output Speed: integer;  
  
  loop  
    var Distance := 0 : integer in  
  
      abort  
        every immediate Metre do  
          Distance := Distance + 1;  
        end every;  
      when Second do  
        emit Speed (Distance);  
      end abort;  
    end var;  
  end loop;  
end module;
```



These are not exclusive



Immediate attribute takes  
every metre into account.



Above block is hard aborted  
with every 'Second' signal



# *Introduction & Languages*

## *Synchronous languages*

### *Causality and Synchronous Languages*

General terms:

‘The future should not influence the past’

Technically:

**Causal synchronous programs** are:

1. **Reactive**  provide a *well-defined output* for each signal sequence.
2. **Deterministic**  provide *exactly one output* for each signal sequence.



# *Introduction & Languages*

## *Synchronous languages*

### *Non-causality in Synchronous Languages*

☞ Non-reactive output:

```
module non-reactive;
  output 0;
  present 0 else emit 0 end;
end module;
```

☞ Non-deterministic output:

```
module non-deterministic;
  output 0;
  present 0 then emit 0 end;
end module;
```

☞ Cyclic dependencies with multiple signals:

```
module cyclic_dependency;
  output A, B;
  [ present A then emit B end || present B else emit A end ]
end module;
```

☞ All examples contain a reference to “the future”, i.e. are “cyclic”.



# *Introduction & Languages*

## *Synchronous languages*

### *Causality in Synchronous Languages*

*Cyclic dependencies can cause causality problems in synchronous languages (similar to potential dead-locks in asynchronous languages).*

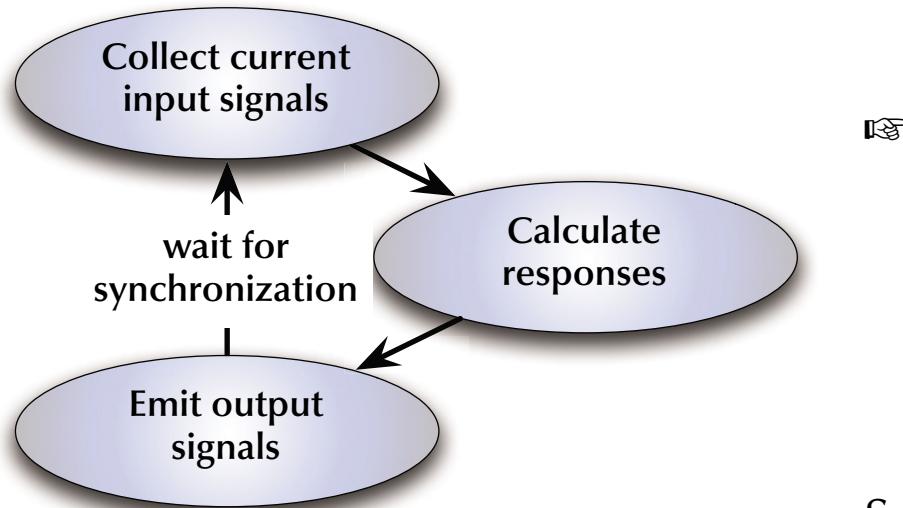
- ☞ **Strict synchronous languages:** avoid all cyclic dependencies in signals.
  - ☞ **Esterel:** fully acyclic programs are considered too restrictive, because cyclic dependencies can make programs more intuitive/simpler.
- ☞ *Cyclic programs can be reactive and deterministic.*



# *Introduction & Languages*

## *Synchronous languages*

### *Strong synchrony or “zero delay” assumption*



The system is assumed '**synchronous**' or '**instantaneous**', iff  
the total worst case computation time is smaller than the minimal time between two observable changes in the environment.

Synchronous systems assume a **logical** or **discrete** rather than continuous time.

Enables:

- ☞ Strong analysis and simplification tools.
- ☞ Significantly easier program verification.
- ☞ Straight forward hardware implementations.



# *Introduction & Languages*

## **VHSIC hardware description language (VHDL)**

### **VHDL**

- **Standardized** hardware description language (IEEE 1076-2008)
- Can be **data-flow** or **control-flow** oriented.
- Programming in the large (**packages**, **generics**).
- **Entities**, **architectures** (processes), and **configurations** are separate.
- Signals can be **digital** or **analog**.
- **Strong typing** (all basic Ada types plus low level types: std\_logic).
- Modules can be **clocked independently** or not at all (**combinatorial**).
- **Concurrency** only limited by number of gates on module (☞ millions).
- **High level synchronization primitives** (protected types).

# VHDL state machine example

```
entity enum is port (Clock, Reset : in Std_Logic;
                     A, B           : in Boolean;
                     Out          : out Std_Logic);
```

```
end enum;
```

```
architecture A_Simple_Moore_Machine of enum is
```

```
type States is (Start, A_Detected, B_Detected, AB_Detected);
```

```
attribute enum_encoding : string;
```

```
attribute enum_encoding of States : type is "one-hot"; -- "grey", "binary", ..
```

```
signal Current_State, Next_State: States;
```

```
begin
```

```
Synchronous_Proc: process (Clock, Reset)
```

```
begin
```

```
if (Reset='1') then
```

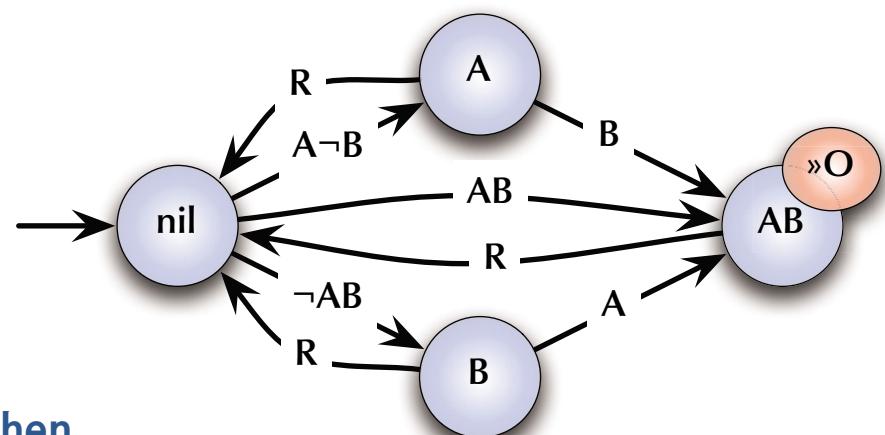
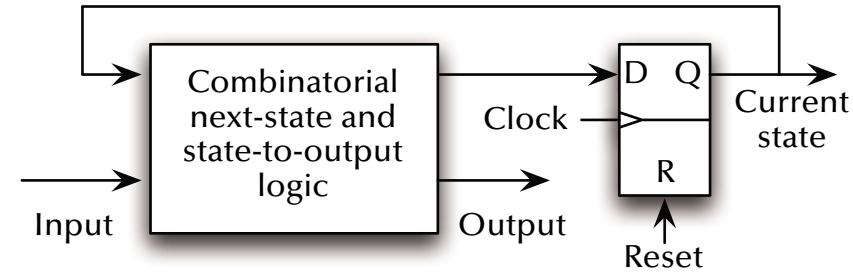
```
    Current_State <= Start;
```

```
elsif (Clock'event and Clock = '1') then
```

```
    Current_State <= Next_State after 1 ns; -- inertial delay
```

```
end if;
```

```
end process; -- End Synchronous_Proc
```



# VHDL state machine example

Combinatorial\_Process: process (Current\_State, A, B)

begin

case Current\_State is

when Start => Out <= '0' after 1 ns;

if A and not B then Next\_State <= A\_Detected;

elsif not A and B then Next\_State <= B\_Detected;

elsif A and B then Next\_State <= AB\_Detected;

else Next\_State <= Start;

end if;

when A\_Detected => Out <= '0' after 1 ns;

if B then Next\_State <= AB\_Detected;

else Next\_State <= A\_Detected;

end if;

when B\_Detected => Out <= '0' after 1 ns;

if A then Next\_State <= AB\_Detected;

else Next\_State <= B\_Detected;

end if;

when AB\_Detected => Out <= '1' after 1 ns;

Next\_State <= AB\_Detected;

end case;

end process; -- End Combinatorial\_Process

end; -- End architecture A\_Simple\_Moore\_Machine



# *Introduction & Languages*

## *Process Algebras*

### *Timed CSP*

$$\begin{aligned} P ::= & \text{Stop} \mid \text{Skip} \mid \text{Wait } t \mid a \rightarrow P \mid P;P \mid P \square P \mid P \sqcap P \mid a:A \rightarrow P_a \mid P \stackrel{t}{\triangleright} P \mid \\ & P \stackrel{t}{\ntriangleleft} P \mid P \triangle P \mid f(P) \mid P \setminus A \mid \|_{A_P} P \mid P_A \|_A P \mid P \parallel P \mid P \|_A P \mid \mu X \cdot F(X) \end{aligned}$$

where:  $P$  is a process,  $a$  is an event,  $A$  is a set of events, and  $t$  is a non-negative real number.

$\text{Stop}$	: Terminal process.	$P \stackrel{t}{\ntriangleleft} P$	: Interrupt first process at time $t$
$\text{Skip}$	: Null process.	$P \triangle P$	: First process until second one starts
$\text{Wait } t$	: Delay process.	$f(P)$	: Relabelling
$a \rightarrow P$	: Process $P$ is preceded by event $a$ .	$P \setminus A$	: Omitting events out of $A$
$P;P$	: Processes in sequence.	$\ _{A_P} P$	: Process set def. by event set
$P \square P$	: Deterministic alternative.	$P_A \ _A P$	: Sync. on overlapping events
$P \sqcap P$	: Non-deterministic alternative.	$P \parallel P$	: Concurrent processes
$a:A \rightarrow P_a$	: Process $P$ is preceded by one event $a \in A$ .	$P \ _A P$	: Sync. on events out of $A$
$P \stackrel{t}{\triangleright} P$	: Alternative based on time	$\mu X \cdot F(X)$	: Recursion



# *Introduction & Languages*

## *Process Algebras*

### *Timed CSP*

Timed CSP (1986 onwards) is an extension of Communication Sequential Processes (CSP, introduced by Hoare in 1978).

- **Everything is a process** constructed via a small set of primitives.
  - Events are **instantaneous**.
  - All communications are **synchronous**.
  - Processes progress until **synchronization points** or terminate.
  - **Unlimited concurrency**.
  - **Traces** denote possible chains of events.
- ☞ **Proofs** that certain traces can / cannot happen are constructed via **algebraic transformations**.

It forms a conceptual / algebraic basis for several concurrent languages incl. Esterel, Ada, Occam, VHDL, Go, VerilogCSP.



# *Introduction & Languages*

## **PEARL**

### ***Process and Experiment Automation Realtime Language***

- Simple and ‘readable’ language for small projects.
- Supports tasking and timed activations.
- Supports interrupts, signals, semaphores, and bolt variables.
- Lacks support for ‘programming in the large’.

Is a settled standard:

- DIN 66253-1: Basic PEARL 1981
- DIN 66253-2: Full PEARL 1982 ↗ both replace by DIN 66253-2: PEARL 90 1998
- DIN 66253-3: PEARL for distributed systems 1989



# *Introduction & Languages*

## **PEARL**

**MODULE**;

**SYSTEM**;

Alert: Hard\_Int (7);

**PROBLEM**;

**SPECIFY** Alert **INTERRUPT**;

**SPECIFY** Help **TASK GLOBAL**;

**SPECIFY** Pushed **BIT GLOBAL**;

**DECLARE** Switch **BIT INITIAL** 0;

Init : **TASK MAIN**;

**WHEN** Alert **ACTIVATE** Recovery;

**ENABLE** Alert;

    Switch := Pushed;

**END**;

Recovery: **TASK PRIORITY** 9;

**DISABLE** Alert;

**IF** Switch = 1 **THEN ACTIVATE** Help; **FIN**;

**AFTER** 30 **MIN ALL** 5 **MIN DURING** 1 **HRS ACTIVATE** Help;

**END**;

**MODEND**;



# *Introduction & Languages*

## **PEARL**

### ***Process and Experiment Automation Realtime Language***

- Established standard.
- Compilers available for all major OSs (and some RT-OSs) as well as for a number of single-board systems (one free compiler for academic users).
- Used for educational purposes mostly.
- A configuration part (“SYSTEM”) allows for hardware migration.
- Synchronization primitives on the level of semaphores.
- Designed for small scale engineering applications.

Currently maintained by a German special-interest community and one company (IEP).



# *Introduction & Languages*

## ***POSIX***

### ***Portable Operating System Interface for Unix***

- IEEE/ANSI Std 1003.1 and following.
- Library Interface (API)  
[C Language calling conventions – types exit mostly in terms of (open) lists of pointers and integers with overloaded meanings].
- More than 30 different POSIX standards (and growing / changing).
  - ☞ a system is ‘POSIX compliant’, if it implements parts of one of them!
  - ☞ a system is ‘100% POSIX compliant’, if it implements one of them!



# *Introduction & Languages*

## ***POSIX - some of the relevant standards...***

1003.1 12/01	<b>OS Definition</b>	single process, multi process, job control, signals, user groups, file system, file attributes, file device management, file locking, device I/O, device-specific control, system database, pipes, FIFO, ...
1003.1b 10/93	<b>Real-time Extensions</b>	real-time signals, priority scheduling, timers, asynchronous I/O, prioritized I/O, synchronized I/O, file sync, mapped files, memory locking, memory protection, message passing, semaphore, ...
1003.1c 6/95	<b>Threads</b>	multiple threads within a process; includes support for: thread control, thread attributes, priority scheduling, mutexes, mutex priority inheritance, mutex priority ceiling, and condition variables
1003.1d 10/99	<b>Additional Real-time Extensions</b>	new process create semantics (spawn), sporadic server scheduling, execution time monitoring of processes and threads, I/O advisory information, timeouts on blocking functions, device control, and interrupt control
1003.1j 1/00	<b>Advanced Real-time Extensions</b>	typed memory, nanosleep improvements, barrier synchronization, reader/writer locks, spin locks, and persistent notification for message queues
1003.21 /-	<b>Distributed Real-time</b>	buffer management, send control blocks, asynchronous and synchronous operations, bounded blocking, message priorities, message labels, and implementation protocols



# *Introduction & Languages*

## ***POSIX - 1003.1b/c***

Frequently employed POSIX features include:

- **Threads:** a common interface to threading - differences to 'classical UNIX processes'
- **Timers:** delivery is accomplished using POSIX signals
- **Priority scheduling:** fixed priority, 32 priority levels
- **Real-time signals:** signals with multiple levels of priority
- **Semaphore:** named semaphore
- **Memory queues:** message passing using named queues
- **Shared memory:** memory regions shared between multiple processes
- **Memory locking:** no virtual memory swapping of physical memory pages



# *Introduction & Languages*

## ***POSIX - support by different operating systems***

	POSIX 1003.1 (Base POSIX)	POSIX 1003.1b (Real-time extensions)	POSIX 1003.1c (Threads)
Solaris	Full support	Full support	Full support
IRIX	Conformant	Full support	Full support
LynxOS	Conformant	Full support	Conformant
QNX Neutrino	Full support	Partial support (no memory locking)	Full support
Linux	Full support	Partial support (no timers, no message queues)	Full support
VxWorks	Partial support (different process model)	Partial support (different process model)	Support through third party add-ons



# *Introduction & Languages*

## ***POSIX - other languages***

POSIX is a 'C' standard...

... but bindings to other languages are also (suggested) POSIX standards:

- **Ada:** 1003.5\*, 1003.24  
(some PAR approved only, some withdrawn, some (partly) implemented)
- **Fortran:** 1003.9 (6/92)
- **Fortran90:** 1003.19 (withdrawn)

... and there are POSIX standards for task-specific **POSIX profiles**, e.g.:

- Super computing: 1003.10 (6/95)
- **Realtime:** 1003.13, 1003.13b (3/98) - profiles 51-54: combinations of the above RT-relevant POSIX standards ↗ RT-Linux
- **Embedded Systems:** 1003.13a (PAR approved only)



# *Introduction & Languages*

## ***POSIX - example: setting a timer***

```
void timer_create(int num_secs, int num_nsecs)
{
    struct sigaction sa;
    struct sigevent sig_spec;
    sigset_t allsigs;
    struct itimerspec tmr_setting;
    timer_t timer_h;

    /* setup signal to respond to timer */
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = timer_intr;

    if (sigaction(SIGRTMIN, &sa, NULL) < 0)
        perror('sigaction');

    sig_spec.sigev_notify = SIGEV_SIGNAL;
    sig_spec.sigev_signo = SIGRTMIN;

    /* create timer, which uses the REALTIME clock */
    if (timer_create(CLOCK_REALTIME, &sig_spec, &timer_h) < 0)
        perror('timer create');
```



# *Introduction & Languages*

## ***POSIX - example: setting a timer (cont.)***

```
/* set the initial expiration and frequency of timer */
tmr_setting.it_value.tv_sec = 1;
tmr_setting.it_value.tv_nsec = 0;
tmr_setting.it_interval.tv_sec = num_secs;
tmr_setting.it_interval.tv_nsec = num_nsecs;

if ( timer_settime(timer_h, 0, &tmr_setting,NULL) < 0)
    perror('settimer');

/* wait for signals */
sigemptyset(&allsigs);
while (1) {
    sigsuspend(&allsigs);
}
/* routine that is called when timer expires */
void timer_intr(int sig, siginfo_t *extra, void *cruft)
{
    /* perform periodic processing and then exit */
}
```



# *Introduction & Languages*

## ***POSIX - example: setting a timer (cont.)***

```
/* set the initial expiration and frequency of timer */
tmr_setting.it_value.tv_sec = 1;
tmr_setting.it_value.tv_nsec = 0;
tmr_setting.it_interval.tv_sec = num_secs;
tmr_setting.it_interval.tv_nsec = num_nsecs;

if ( timer_settime(timer_h, 0, &tmr_setting,NULL) < 0)
    perror('settimer');

/* wait for signals */
sigemptyset(&allsigs);
while (1) {
    sigsuspend(&allsigs);
}
/* routine that is called when timer expires */
void timer_intr(int sig, siginfo_t *extra, void *cruft)
{
    /* perform periodic processing and then exit */
}
```

Compare to Pearl:  
AFTER 30 MIN ALL 5 MIN DURING 1 HRS ACTIVATE Help;



# *Introduction & Languages*

## *Assembler level programming*

### *Macro-Assemblers*

- ☞ Closest to hardware.
- ☞ Predictable results (as predictable as the underlying hardware).
- ☞ Small footprint.
- ☞ As sequential or concurrent as the underlying hardware.
- No abstraction or support for large systems.
- Basic types are defined by the deployed processor (similar to C).
- Hard to read.
- ☞ Used mostly in very small applications or short code sequences.



# *Introduction & Languages*

## ***Cross-over between assembler and higher level languages***

### **C**

Combines the main disadvantages of assemblers and higher programming languages:

- ⚡ Does not offer the abstractions of a higher programming language.
- ⚡ Cannot take direct advantage of hardware-specific features.

Yet:

- ☞ Extremely popular.
- ☞ Simple and fast parameter passing (without compiler optimizations).
- ☞ Small footprint (zero runtime environment).
- ☞ Simple to write compilers (basically a macro-assembler).
- ☞ Available for virtually any processor.



# *Introduction & Languages*

## *Real-Time Programming Languages*

### *Languages mentioned so far*

- Ada (Ada2012) ↗ General workhorse.
- Real-Time Java (Real-Time Specification for Java 1.1) ↗ (Very) soft real-time applications.
- Esterel (Esterel v7) ↗ An alternative for high-integrity applications.
- VHDL ↗ Compile real-time data flows and independent, asynchronous control paths into hardware.
- Timed CSP (as used and developed since 1986) ↗ An algebraic approach.
- PEARL (PEARL-90) ↗ A traditional language, specialized on plant modelling.
- POSIX (POSIX 1003.1b, ...) ↗ The libraries of bare bone integers and semaphores.
- Assemblers / C ↗ The languages of bare bone words.



# *Introduction & Languages*

## *Summary*

# *Introduction & Real-Time Languages*

- **Features** (and non-features) of a real-time system
  - Features, definitions, scenarios, and characteristics.
- **Components** of a real-time system
  - Converters, interfaces, sensors, actuators, communication systems, controllers, ...
- **Software layers** of a real-time system
  - Algorithms, operating systems, protocols, languages, concurrent and distributed systems.
- **Real-time languages criteria**
  - Mostly high integrity, predictable languages with means for explicit time scopes.
- **Examples of actual real-time languages**