Systems, Networks & Concurrency 2020

Language refresher / introduction course

Uwe R. Zimmer - The Australian National University

Language refresher / introduction course

References for this chapter

[Ada 2012 Language Reference Manual]
see course pages or http://www.ada-auth.org/standards/ada12.html

[Chapel 1.13 Language Specification Version 0.981]
see course pages or
http://chapel.cray.com/docs/latest/_downloads/chapelLanguageSpec.pdf
released on 7. April 2016

Language refresher / introduction course

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), abstract data types, generics, 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, informations systems, safety and security issues.

Language refresher / introduction course

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**')
- **Abstract** types and **dispatching**

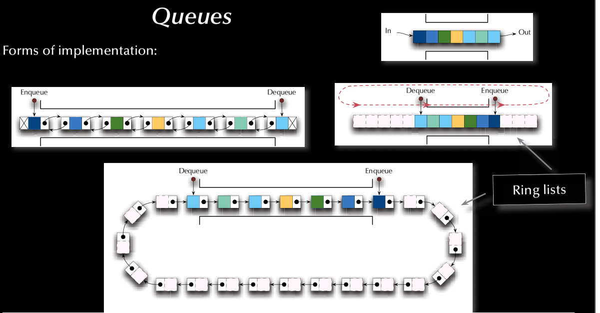
Not mentioned here: general object orientation, dynamic memory management, foreign language interfaces, marshalling, basics of imperative programming, …

Language refresher / introduction course

Data structure example

Queues

Forms of implementation:

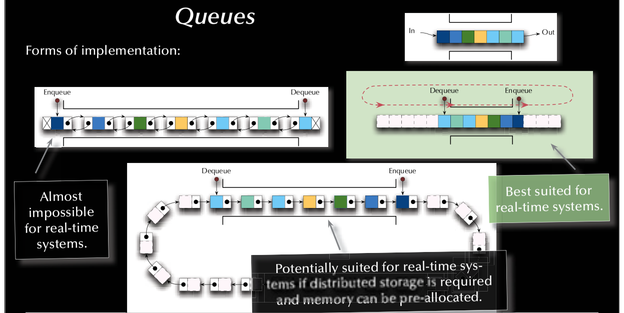


Language refresher / introduction course

Data structure example

Queues

Forms of implementation:



Language refresher / introduction course

Ada

Basics

… introducing:

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

Language refresher / introduction course

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;
```


Language refresher / introduction course

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.

29

Language refresher / introduction course

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.

30

Language refresher / introduction course

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, predicable values**, e.g. 1_000

31

Language refresher / introduction course

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!

32

Language refresher / introduction course

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!

33

Language refresher / introduction course

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'.

34

Language refresher / introduction course

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).

35

Language refresher / introduction course

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?

36

Language refresher / introduction course

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;
```


37

Language refresher / introduction course

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;
```

Implementations are defined in a **separate** file.
Syntactically enclosed in a package body block.

38

Language refresher / introduction course

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.

39

Language refresher / introduction course

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

40

Language refresher / introduction course

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 without begin-end blocks.

41

Language refresher / introduction course

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?

42

Language refresher / introduction course

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;
```


43

Language refresher / introduction course

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.

44

Language refresher / introduction course

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.

45

Language refresher / introduction course

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`".

46

Language refresher / introduction course

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 …

47

Language refresher / introduction course

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?

48

Language refresher / introduction course

Ada

Exceptions

… introducing:

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

49

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;
```

50

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).

51

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.

52

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?

53

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;
```


54

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.

55

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.

56

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?

57

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;
```

58

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.

59

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?

60

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.

61

Language refresher / introduction course

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

62

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;
```

63

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.

64

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.

65

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.

66

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?

67

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 Is_Full (Queue) 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 Is_Empty (Queue) 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;
```

Identical

68

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 Is_Full (Queue) 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 Is_Empty (Queue) 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;
```

Identical

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

69

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 Is_Full (Queue) 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 Is_Empty (Queue) 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;
```

Identical

… anything on this slide still not perfectly clear?

70

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;
```

71

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.

72

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.)

73

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?

74

Language refresher / introduction course

Ada

Contracts

… introducing:

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

75

A contracting queue specification

```ada
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;
```

76

A contracting queue specification

```ada
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 Bo
   function Is_Full   (Q : Queue_Type) return Bo
   function Length    (Q : Queue_Type) return Na
   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.

> ∀ and ∃ quantifiers are expressed as "for all" and "for some" expressions resp.

77

A contracting queue specification

> … anything on this slide still not perfectly clear?

```ada
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;
```

78

A contracting queue specification (cont.)

```ada
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;
```

79

A contracting queue specification (cont.)

```ada
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.

80

A contracting queue specification (cont.)

> … anything on this slide still not perfectly clear?

```ada
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;
```

81

A contracting queue implementation

Identical

```ada
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.

82

A contracting queue test program

```ada
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 (Is_Empty (Queue)));
exception
   when Exception_Id : Assert_Failure => Show_Exception (Exception_Id);
end Queue_Test_Contract;
```

83

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 (Is_Empty (Queue)));
exception
   when Exception_Id : Assert_Failure => Show_Exception (Exception_Id);
end Queue_Test_Contract;
```

Violated Pre-condition will raise an assert failure exception.

84

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 (Is_Empty (Queue)));
exception
   when Exception_Id : Assert_Failure => Show_Exception (Exception_Id);
end Queue_Test_Contract;
```

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

85

A contracted queue specification

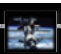
```
package Queue_Pack_Contract is
  (…)
  procedure Enqueue (Item : Element; Q : in out Queue_Type) with
    Pre  => not Is_Full  (Q), -- could also be "=> True" according to specifications
    Post => not Is_Empty (Q) and then Length (Q) = Length (Q'Old) + 1
              and then Lookahead (Q, Length (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;

   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);
  (…)
```

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

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

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.

86



Language refresher / introduction course

Ada

Generic (polymorphic) packages

... introducing:

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

87

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;
```

88

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
```

89

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

90

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?

91

A generic queue implementation

```
package body Queue_Pack_Generic is
   procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
   begin
      if Is_Full (Queue) then
         raise Queueoverflow;
      end if;
      Queue.Elements (Queue.Free) := Item;
      Queue.Free                  := [illegible] (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 Queueunderflow; end if;
      Item           := Queue.Elements (Queue.Top);
      Queue.Top      := [illegible] (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;
```

Identical

92

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;
```

93

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

94

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?

95

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.

96

Language refresher / introduction course

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**

97

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;
```


98

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.

99

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 a access routines are distinguished by the keywords: entry, procedure, function

100

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.

101

A generic protected queue specification

```ada
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.

102

A generic protected queue specification

```ada
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.

103

A generic protected queue specification

```ada
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?

104

A generic protected queue implementation

```ada
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;
```


105

A generic protected queue implementation

```ada
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;
      ...
      end Empty_Queue;
      function Is_Empty ...
      function Is_Full ...
        (not Queue.Is_E...
   end Protected_Queue;
end Queue_Pack_Protected...
```

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.

106

A generic protected queue implementation

```ada
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?

107

A generic protected queue test program

```ada
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;
```


108

A generic protected queue test program

```ada
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.

109

A generic protected queue test program

```ada
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.

110

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.

111

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?

112

A generic protected queue test program

```
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;
```


113

A generic protected queue test program

```
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.

114

A generic protected queue test program

```
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.

115

A generic protected queue test program

```
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 once all inner tasks are terminated).

116

A generic protected queue test program

```
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?

117

A generic protected queue test program

```
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;
```


118

A generic protected queue test program

```
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.

119

A generic protected queue test program

```
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?

120

A generic protected queue test program

```
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?

121

A generic protected queue test program

```
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?

122

Language refresher / introduction course

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.**

123

Language refresher / introduction course

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!

124

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;
```

125

An abstract queue specification

Motivation:

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

```
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;
```

126

An abstract queue specification

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.

```
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;
```

127

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.

128

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?

129

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);
      not overriding procedure Empty_Queue;
      not overriding function Is_Empty return Boolean;
      not overriding function Is_Full  return Boolean;
   private
      Queue : Queue_Type;
   end Protected_Queue;
private
   (...) -- as all previous private queue declarations
end Queue_Pack_Concrete;
```

130

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.

131

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.

132

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);
      not overriding procedure Empty_Queue;
      not overriding function Is_Empty return Boolean;
      not overriding function Is_Full  return Boolean;
   private
      Queue : Queue_Type;
   end Protected_Queue;
private
   (...) -- as all previous private queue declarations
end Queue_Pack_Concrete;
```

Other (not-overriding) methods can be added.

133

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?

134

A concrete queue implementation

Identical

```
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;
```


135

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;
```


136

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;
(...)
begin
   null;
end Queue_Test_Dispatching;
```

Declaring two concrete tasks.
(Queue_User has a synchronous message passing entry)

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);
      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;
```

Declaring local queues in each task.

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'

146

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?

147

Language refresher / introduction course

Ada

Ada language status


Boeing 787 cockpit (press release photo)

- 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.

☞ 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

148

Language refresher / introduction course

Chapel

Currently under development at Cray.
(originally for the DARPA High Productivity Computing Systems initiative.)

☞ Targeted at massively parallel computers

Language primitives for ...

- Data parallelism:
  - ☞ Distributed data storage with fine grained control ("domains").
  - ☞ Concurrent map operations (`forall`).
  - ☞ Concurrent fold operations (`scan, reduce`).
- Task parallelism:
  - ☞ concurrent loops and blocks (`cobegin, coforall`).
- Synchronization:
  - ☞ Task synchronization, synchronized variables, atomic sections.

149

A data-parallel stencil program

```
config const n              = 100,
             max_iterations = 50,
             epsilon        = 1.0E-5,
             initial_border = 1.0;

const Matrix_w_Borders = {0 .. n + 1, 0 .. n + 1, 0 .. n + 1},
      Matrix           = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
      Single_Border    = Matrix.exterior (1, 0, 0);

var Field      : [Matrix_w_Borders] real,
    Next_Field : [Matrix]           real;

proc Stencil (M : [/* Matrix_w_Borders */] real, (i, j, k) : index (Matrix)) : real {
  return (M [i - 1, j, k]
        + M [i + 1, j, k]
        + M [i, j - 1, k]
        + M [i, j + 1, k]
        + M [i, j, k + 1]
        + M [i, j, k - 1]) / 6;
}
```

150

A data-parallel stencil program

```
config const n              = 100,
             max_iterations = 50,
             epsilon        = 1.0E-5,
             initial_border = 1.0;

const Matrix_w_Borders = {0 .. n + 1, 0 .. n + 1, 0 .. n + 1},
      Matrix           = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
      Single_Border    = Matrix.exterior (1, 0, 0);

var Field      : [Matrix_w_Borders] real,
    Next_Field : [Matrix]           real;

proc Stencil (M : [/* Matrix_w_Borders */] real, (i, j, k) : index (Matrix)) : real {
  return (M [i - 1, j, k]
        + M [i + 1, j, k]
        + M [i, j - 1, k]
        + M [i, j + 1, k]
        + M [i, j, k + 1]
        + M [i, j, k - 1]) / 6;
}
```

Configuration constants can be set via command line options: `./Stencil --n=500`

151

A data-parallel stencil program

```
config const n              = 100,
             max_iterations = 50,
             epsilon        = 1.0E-5,
             initial_border = 1.0;

const Matrix_w_Borders = {0 .. n + 1, 0 .. n + 1, 0 .. n + 1},
      Matrix           = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
      Single_Border    = Matrix.exterior (1, 0, 0);

var Field      : [Matrix_w_Borders] real,
    Next_Field : [Matrix]           real;

proc Stencil (M : [/* Matrix_w_Borders */] real, (i, j, k) : index (Matrix)) : real {
  return (M [i - 1, j, k]
        + M [i + 1, j, k]
        + M [i, j - 1, k]
        + M [i, j + 1, k]
        + M [i, j, k + 1]
        + M [i, j, k - 1]) / 6;
}
```

Defining **domains** to be used for multi-dimensional array declarations and assignments.

152

A data-parallel stencil program

```
config const n              = 100,
             max_iterations = 50,
             epsilon        = 1.0E-5,
             initial_border = 1.0;

const Matrix_w_Borders = {0 .. n + 1, 0 .. n + 1, 0 .. n + 1},
      Matrix           = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
      Single_Border    = Matrix.exterior (1, 0, 0);

var Field      : [Matrix_w_Borders] real,
    Next_Field : [Matrix]           real;

proc Stencil (M : [/* Matrix_w_Borders */] real, (i, j, k) : index (Matrix)) : real {
  return (M [i - 1, j, k]
        + M [i + 1, j, k]
        + M [i, j - 1, k]
        + M [i, j + 1, k]
        + M [i, j, k + 1]
        + M [i, j, k - 1]) / 6;
}
```

Declaring matrices of different, yet related dimensions.

153

A data-parallel stencil program

```
config const n              = 100,
             max_iterations = 50,
             epsilon        = 1.0E-5,
             initial_border = 1.0;

const Matrix_w_Borders = {0 .. n + 1, 0 .. n + 1, 0 .. n + 1},
      Matrix           = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
      Single_Border    = Matrix.exterior (1, 0, 0);

var Field      : [Matrix_w_Borders] real,
    Next_Field : [Matrix]           real;

proc Stencil (M : [/* Matrix_w_Borders */] real, (i, j, k) : index (Matrix)) : real {
  return (M [i - 1, j, k]
        + M [i + 1, j, k]
        + M [i, j - 1, k]
        + M [i, j + 1, k]
        + M [i, j, k + 1]
        + M [i, j, k - 1]) / 6;
}
```

Note the index type

Function which calculates a "stencil" value at a spot inside a given matrix

154

A data-parallel stencil program

```
config const n              = 100,
             max_iterations = 50,
             epsilon        = 1.0E-5,
             initial_border = 1.0;

const Matrix_w_Borders = {0 .. n + 1, 0 .. n + 1, 0 .. n + 1},
      Matrix           = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
      Single_Border    = Matrix.exterior (1, 0, 0);

var Field      : [Matrix_w_Borders] real,
    Next_Field : [Matrix]           real;

proc Stencil (M : [/* Matrix_w_Borders */] real, (i, j, k) : index (Matrix)) : real {
  return (M [i - 1, j, k]
        + M [i + 1, j, k]
        + M [i, j - 1, k]
        + M [i, j + 1, k]
        + M [i, j, k + 1]
        + M [i, j, k - 1]) / 6;
}
```

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

155

A data-parallel stencil program (cont.)

```
Field [Single_Border] = initial_border;

for l in 1 .. max_iterations {

  forall Matrix_Indices in Matrix do
    Next_Field (Matrix_Indices) = Stencil (Field, Matrix_Indices);

  const delta = max reduce abs (Field [Matrix] - Next_Field);

  Field [Matrix] = Next_Field;

  if delta < epsilon then break;
}
```

156

A data-parallel stencil program (cont.)

```
Field [Single_Border] = initial_border;

for l in 1 .. max_iterations {

  forall Matrix_Indices in Matrix do
    Next_Field (Matrix_Indices) = Stencil (Field, Matrix_Indices);

  const delta = max reduce abs (Field [Matrix] - Next_Field);

  Field [Matrix] = Next_Field;

  if delta < epsilon then break;
}
```

Scalar to 2-d array-slice assignment
(Technically a 3-d domain with
two degenerate dimensions)

3-d array to 3-d array-slice assignment

157

A data-parallel stencil program (cont.)

```
Field [Single_Border] = initial_border;

for l in 1 .. max_iterations {

  forall Matrix_Indices in Matrix do
    Next_Field (Matrix_Indices) = Stencil (Field, Matrix_Indices);

  const delta = max reduce abs (Field [Matrix] - Next_Field);

  Field [Matrix] = Next_Field;

  if delta < epsilon then break;
}
```

Data parallel application
of the `Stencil` function
to the whole 3-d matrix

158

A data-parallel stencil program (cont.)

```
Field [Single_Border] = initial_border;

for l in 1 .. max_iterations {

  forall Matrix_Indices in Matrix do
    Next_Field (Matrix_Indices) = Stencil (Field, Matrix_Indices);

  const delta = max reduce abs (Field [Matrix] - Next_Field);

  Field [Matrix] = Next_Field;

  if delta < epsilon then break;
}
```

Data parallel (divide-and-conquer)
application of the `max` function to
the component-wise differences.

"3-d data-parallel version" of (Haskell):

```
foldr max minBound $ zipWith (-) field next_field
```

159

A data-parallel stencil program (cont.)

```
Field [Single_Border] = initial_border;

for l in 1 .. max_iterations {

  forall Matrix_Indices in Matrix do
    Next_Field (Matrix_Indices) = Stencil (Field, Matrix_Indices);

  const delta = max reduce abs (Field [Matrix] - Next_Field);

  Field [Matrix] = Next_Field;

  if delta < epsilon then break;
}
```

… anything on this slide
still not perfectly clear?

160

Language refresher / introduction course

Summary

Language refresher / introduction course

- Specification and implementation (body) parts, basic types
- Exceptions & Contracts
- Information hiding in specifications ('private')
- Generic programming
- Tasking
- Monitors and synchronisation ('protected', 'entries', 'selects', 'accepts')
- Abstract types and dispatching
- Data parallel operations