

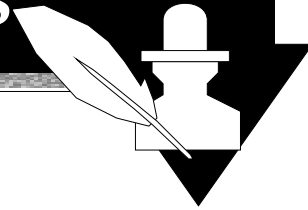


Ada refresher course

*Uwe R. Zimmer & Alistair Rendell
The Australian National University*



Concurrent & Distributed Systems



References for this chapter

[Cohen96]

Norman H. Cohen

Ada as a second language

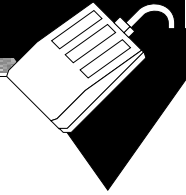
McGraw-Hill series in computer science, 2nd
edition

[Ada 95 Reference manual]

(see lab pages or web)



Concurrent & Distributed Systems



Ada95

Ada95 is a **standardized** (ISO/IEC 8652:1995(E)) 'general purpose' language with **core** language primitives for

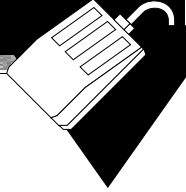
- strong typing, separate compilation (specification and implementation), object-orientation,
- concurrency, monitors, rpcs, timeouts, scheduling, priority ceiling locks
- strong run-time environments

... and **standardized** language-**annexes** for

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



Concurrent & Distributed Systems



Ada95

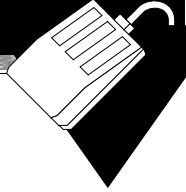
A crash course

... refreshing:

- specification and implementation (body) parts, basic types
- exceptions
- information hiding in specifications ('private')
- generic programming
- class-wide programming ('tagged types')
- monitors and synchronisation ('protected', 'entries', 'selects', 'accepts')
- abstract types and dispatching



Concurrent & Distributed Systems



Ada95

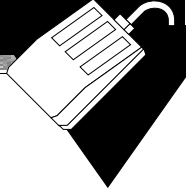
Basics

... introducing:

- specification and implementation (body) parts
- constants
- some basic types (integer specifics)
- some type attributes
- parameter specification



Concurrent & Distributed Systems



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'Range) of Element;

type Queue_Type is **record**

 Top, Free : Marker := Marker'First;

 Elements : List;

end record;

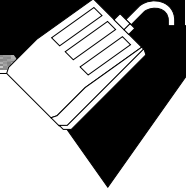
procedure Enqueue (Item: **in** Element; Queue: **in out** Queue_Type);

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

end Queue_Pack_Simple;



Concurrent & Distributed Systems



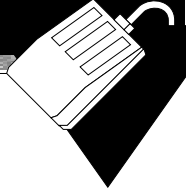
A simple queue *implementation*

```
package body Queue_Pack_Simple is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free - 1;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top - 1;
  end Dequeue;
end Queue_Pack_Simple;
```



Concurrent & Distributed Systems

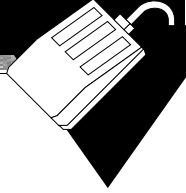


A simple queue test program

```
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); -- will produce an unpredictable result!
end Queue_Test_Simple;
```



Concurrent & Distributed Systems



Ada95

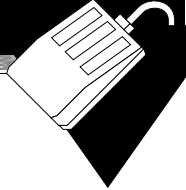
Exceptions

... introducing:

- exception handling
- enumeration types
- functional type attributes



Concurrent & Distributed Systems



A queue *specification* with proper exceptions

```
package Queue_Pack_Exceptions is
```

```
    QueueSize : constant Integer := 10;
```

```
    type Element is (Up, Down, Spin, Turn);
```

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

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

```
    type Queue_State is (Empty, Filled);
```

```
    type Queue_Type is record
```

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

```
        State     : Queue_State := Empty;
```

```
        Elements  : List;
```

```
    end record;
```

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

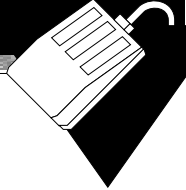
```
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
```

```
    Queueoverflow, Queueunderflow : exception;
```

```
end Queue_Pack_Exceptions;
```



Concurrent & Distributed Systems



A queue *implementations* with proper exceptions

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

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

end Queue_Pack_Exceptions;
```



Concurrent & Distributed Systems



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;

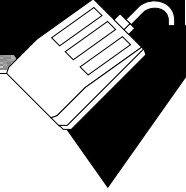
begin
    Enqueue (Turn, Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue); -- will produce a 'Queue underflow'

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

end Queue_Test_Exceptions;
```



Concurrent & Distributed Systems



Ada95

Information hiding (private parts)

... introducing:

- private ➡ assignments and comparisons are allowed
- limited private ➡ entity cannot be assigned or compared



Concurrent & Distributed Systems



*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: in Element; Queue: in out Queue_Type);
```

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

```
  Queueoverflow, Queueunderflow : exception;
```

```
private
```

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

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

```
  type Queue_State is (Empty, Filled);
```

```
  type Queue_Type is record
```

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

```
    State      : Queue_State := Empty;
```

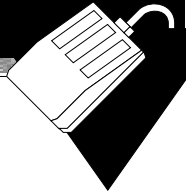
```
    Elements  : List;
```

```
  end record;
```

```
end Queue_Pack_Private;
```



Concurrent & Distributed Systems



A queue *implementations* with proper information hiding

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

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

end Queue_Pack_Private;
```



Concurrent & Distributed Systems



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

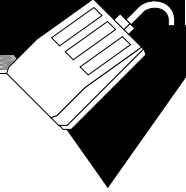
    Queue, Queue_Copy : Queue_Type;
    Item                : Element;

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); -- will produce a 'Queue underflow'

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



Concurrent & Distributed Systems



Ada95

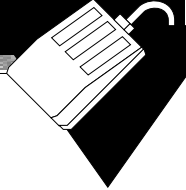
Generic packages

... introducing:

- specification of generic packages
- instantiation of generic packages



Concurrent & Distributed Systems

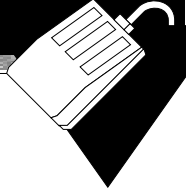


A generic queue *specification*

```
generic
  type Element is private;
package Queue_Pack_Generic is
  QueueSize: constant Integer := 10;
  type Queue_Type is limited private;
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  Queueoverflow, Queueunderflow : exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is record
    Top, Free : Marker      := Marker'First;
    State      : Queue_State := Empty;
    Elements   : List;
  end record;
end Queue_Pack_Generic;
```



Concurrent & Distributed Systems



A generic queue *implementation*

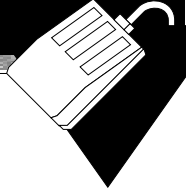
```
package body Queue_Pack_Generic is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free - 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;

end Queue_Pack_Generic;
```



Concurrent & Distributed Systems



A generic queue test program

```
with Queue_Pack_Generic;
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;

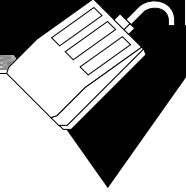
  Queue : Queue_Type;
  Item   : Positive;

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;
```



Concurrent & Distributed Systems



Ada95

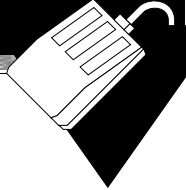
Object oriented programming I

... introducing:

- tagged types → the Ada-way to say that this type can be extended
- derivation of tagged types
- method overwriting
- usage of parent entities



Concurrent & Distributed Systems



*An open queue base class **specification***

```
package Queue_Pack_Object_Base is
  QueueSize : constant Integer := 10;
  type Element is new Positive range 1..1000;
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is tagged record
    Top, Free : Marker      := Marker'First;
    State      : Queue_State := Empty;
    Elements   : List;
  end record;

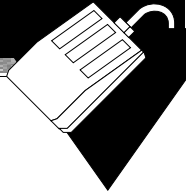
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

  Queueoverflow, Queueunderflow : exception;

end Queue_Pack_Object_Base;
```



Concurrent & Distributed Systems



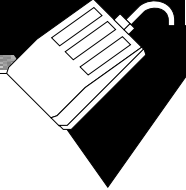
An open queue base class *implementation*

```
package body Queue_Pack_Object_Base is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free - 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;
end Queue_Pack_Object_Base;
```



Concurrent & Distributed Systems



*A derived open queue class **specification***

```
with Queue_Pack_Object_Base; use Queue_Pack_Object_Base;
package Queue_Pack_Object is
```

```
  type Ext_Queue_Type is new Queue_Type with record
```

```
    Reader      : Marker      := Marker'First;
```

```
    Reader_State : Queue_State := Empty;
```

```
  end record;
```

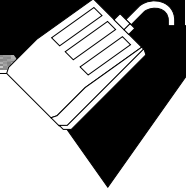
```
  procedure Enqueue      (Item: in Element; Queue: in out Ext_Queue_Type);
```

```
  procedure Read_Queue (Item: out Element; Queue: in out Ext_Queue_Type);
```

```
end Queue_Pack_Object;
```



Concurrent & Distributed Systems



*A derived open queue class **implementation***

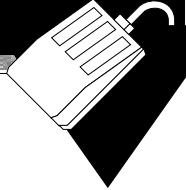
```
package body Queue_Pack_Object is
  procedure Enqueue (Item: in Element; Queue: in out Ext_Queue_Type) is
  begin
    Enqueue (Item, Queue_Type (Queue));
    Queue.Reader_State := Filled;
  end Enqueue;

  procedure Read_Queue (Item: out Element; Queue: in out Ext_Queue_Type) is
  begin
    if Queue.Reader_State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Reader);
    Queue.Reader := Queue.Reader - 1;
    if Queue.Reader = Queue.Free then Queue.Reader_State := Empty; end if;
  end Read_Queue;

end Queue_Pack_Object;
```



Concurrent & Distributed Systems



An open class test program

```
with Queue_Pack_Object_Base; use Queue_Pack_Object_Base;
with Queue_Pack_Object;      use Queue_Pack_Object;
with Ada.Text_IO;            use Ada.Text_IO;

procedure Queue_Test_Object is

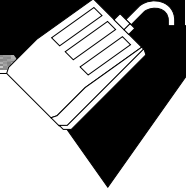
    Queue : Ext_Queue_Type;
    Item   : Element;

begin
    Enqueue (Item => 1, Queue => Queue);
    Read_Queue (Item, Queue);
    Enqueue (Item => 5, Queue => Queue);
    Dequeue (Item, 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_Object;
```



Concurrent & Distributed Systems



Ada95

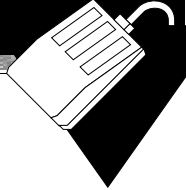
Object oriented programming II

... introducing:

- private tagged types
- objects which are protected against their children also



Concurrent & Distributed Systems



*An encapsulated queue base class **specification***

```
package Queue_Pack_Object_Base_Private is
  QueueSize : constant Integer := 10;
  type Element is new Positive range 1..1000;
  type Queue_Type is tagged limited private;

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

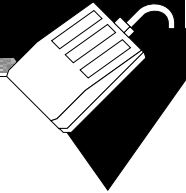
  Queueoverflow, Queueunderflow : exception;

private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is tagged limited record
    Top, Free : Marker      := Marker'First;
    State      : Queue_State := Empty;
    Elements   : List;
  end record;

end Queue_Pack_Object_Base_Private;
```



Concurrent & Distributed Systems



An encapsulated queue base class *implementation*

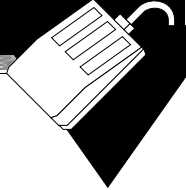
```
package body Queue_Pack_Object_Base_Private is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free - 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;

end Queue_Pack_Object_Base_Private;
```



Concurrent & Distributed Systems

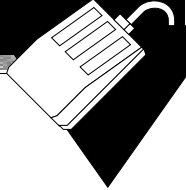


*A derived encapsulated queue class **specification***

```
with Queue_Pack_Object_Base_Private; use Queue_Pack_Object_Base_Private;
package Queue_Pack_Object_Private is
    type Ext_Queue_Type is new Queue_Type with private;
    subtype Depth_Type is Positive range 1..QueueSize;
    procedure Look_Ahead (Item: out Element;
                          Depth: in Depth_Type; Queue: in out Ext_Queue_Type);
private
    type Ext_Queue_Type is new Queue_Type with null record;
end Queue_Pack_Object_Private;
```



Concurrent & Distributed Systems



*A derived encapsulated queue class **implementation***

```
package body Queue_Pack_Object_Private is
```

```
  procedure Look_Ahead (Item: out Element;
```

```
                       Depth: in Depth_Type; Queue: in out Ext_Queue_Type) is
```

```
    Storage      : Queue_Type;
```

```
    ShuffleItem  : Element;
```

```
begin
```

```
  for I in 1..Depth - 1 loop
```

```
    Dequeue (ShuffleItem, Queue);
```

```
    Enqueue (ShuffleItem, Storage);
```

```
  end loop;
```

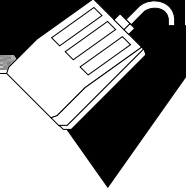
```
  Dequeue (Item, Queue);
```

```
  Enqueue (Item, Storage);
```

```
(...)
```



Concurrent & Distributed Systems



(...)

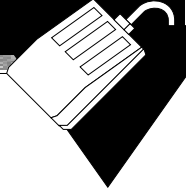
```
Read_The_Rest:
  begin
    for I in 1..QueueSize - Depth loop
      Dequeue (ShuffleItem, Queue);
      Enqueue (ShuffleItem, Storage);
    end loop;
  exception
    when Queueunderflow => null; -- read the rest is done
  end Read_The_Rest;
Restore_The_Queue:
  begin
    for I in 1..QueueSize loop
      Dequeue (ShuffleItem, Storage);
      Enqueue (ShuffleItem, Queue);
    end loop;
  exception
    when Queueunderflow => null; -- restore is done
  end Restore_The_Queue;

end Look_Ahead;

end Queue_Pack_Object_Private;
```



Concurrent & Distributed Systems



An encapsulated class test program

```
with Queue_Pack_Object_Base_Private; use Queue_Pack_Object_Base_Private;
with Queue_Pack_Object_Private;     use Queue_Pack_Object_Private;
with Ada.Text_IO;                   use Ada.Text_IO;

procedure Queue_Test_Object_Private is

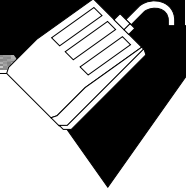
    Queue : Ext_Queue_Type;
    Item   : Element;

begin
    Enqueue (Item => 1, Queue => Queue);
    Enqueue (Item => 1, Queue => Queue);
    Look_Ahead (Item => Item, Depth => 2, Queue => Queue);
    Enqueue (Item => 5, Queue => Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, 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_Object_Private;
```



Concurrent & Distributed Systems



Ada95

Tasks & Monitors

... introducing:

- protected types
- tasks (definition, instantiation and termination)
- task synchronisation
- entry guards
- entry calls
- accept and selected accept statements

A protected queue specification

Package Queue_Pack_Protected is

```
QueueSize : constant Integer := 10;  
subtype Element is Character;  
type Queue_Type is limited private;
```

Protected type Protected_Queue is

```
    entry Enqueue (Item: in Element);  
    entry Dequeue (Item: out Element);
```

private

```
    Queue : Queue_Type;
```

```
end Protected_Queue;
```

private

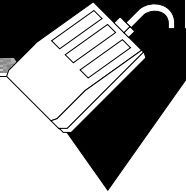
```
type Marker is mod QueueSize;  
type List is array (Marker'Range) of Element;  
type Queue_State is (Empty, Filled);  
type Queue_Type is record  
    Top, Free : Marker      := Marker'First;  
    State      : Queue_State := Empty;  
    Elements  : List;
```

```
end record;
```

```
end Queue_Pack_Protected;
```



Concurrent & Distributed Systems



A protected queue *implementation*

```
package body Queue_Pack_Protected is
  protected body Protected_Queue is
    entry Enqueue (Item: in Element) when
      Queue.State = Empty or Queue.Top /= Queue.Free is
    begin
      Queue.Elements (Queue.Free) := Item;
      Queue.Free := Queue.Free - 1;
      Queue.State := Filled;
    end Enqueue;

    entry Dequeue (Item: out Element) when
      Queue.State = Filled is
    begin
      Item := Queue.Elements (Queue.Top);
      Queue.Top := Queue.Top - 1;
      if Queue.Top = Queue.Free then Queue.State := Empty; end if;
    end Dequeue;

  end Protected_Queue;
end Queue_Pack_Protected;
```

A multitasking protected queue test program

```
with Queue_Pack_Protected; use Queue_Pack_Protected;
with Ada.Text_IO;         use Ada.Text_IO;

procedure Queue_Test_Protected is

  Queue : Protected_Queue;

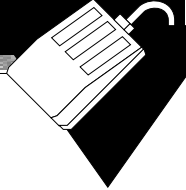
  task Producer is entry shutdown; end Producer;
  task Consumer is          end Consumer;

  task body Producer is
    Item   : Element;
    Got_It : Boolean;
  begin
    loop
      select
        accept shutdown; exit; -- main task loop
      else
        Get_Immediate (Item, Got_It);
        if Got_It then
          Queue.Enqueue (Item); -- task might be blocked here!
        else
          delay 0.1; --sec.
        end if;
      end select;
    end loop;
  end Producer;
```

(...)



Concurrent & Distributed Systems



A multitasking protected queue test program (cont.)

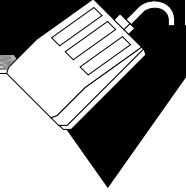
(...)

```
task body Consumer is
  Item : Element;
begin
  loop
    Queue.Dequeue (Item); -- task might be blocked here!
    Put ("Received: "); Put (Item); Put_Line ("!");
    if Item = 'q' then
      Put_Line ("Shutting down producer"); Producer.Shutdown;
      Put_Line ("Shutting down consumer"); exit; -- main task loop
    end if;
  end loop;
end Consumer;

begin
  null;
end Queue_Test_Protected;
```



Concurrent & Distributed Systems



Ada95

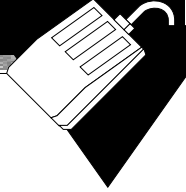
Abstract types & dispatching

... introducing:

- abstract tagged types
- abstract subroutines
- concrete implementation of abstract types
- dispatching to different packages, tasks, and partitions according to concrete types



Concurrent & Distributed Systems

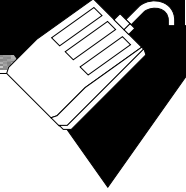


*An abstract queue **specification***

```
package Queue_Pack_Abstract is
  subtype Element is Character;
  type Queue_Type is abstract tagged limited private;
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
    abstract;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
    abstract;
private
  type Queue_Type is abstract tagged limited null record;
end Queue_Pack_Abstract;
```



Concurrent & Distributed Systems



A concrete queue *specification*

```
with Queue_Pack_Abstract; use Queue_Pack_Abstract;
package Queue_Pack_Concrete is
```

```
    QueueSize : constant Integer := 10;
    type Real_Queue is new Queue_Type with private;

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

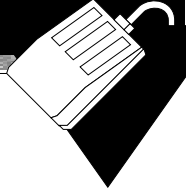
    Queueoverflow, Queueunderflow : exception;
```

```
private
```

```
    type Marker is mod QueueSize;
    type List is array (Marker'Range) of Element;
    type Queue_State is (Empty, Filled);
    type Real_Queue is new Queue_Type with record
        Top, Free : Marker      := Marker'First;
        State      : Queue_State := Empty;
        Elements   : List;
    end record;
end Queue_Pack_Concrete;
```



Concurrent & Distributed Systems



A concrete queue *implementation*

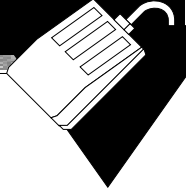
```
package body Queue_Pack_Concrete is
  procedure Enqueue (Item: in Element; Queue: in out Real_Queue) is
  begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free - 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Real_Queue) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top - 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;

end Queue_Pack_Concrete;
```



Concurrent & Distributed Systems



A multitasking dispatching test program

```
with Queue_Pack_Abstract; use Queue_Pack_Abstract;
with Queue_Pack_Concrete; use Queue_Pack_Concrete;

procedure Queue_Test_Dispatching is

  type Queue_Class is access all Queue_Type'class;

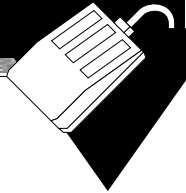
  task Queue_Holder is -- could be on an individual partition
    entry Queue_Filled;
  end Queue_Holder;

  task Queue_User is -- could be on an individual partition
    entry Send_Queue (Remote_Queue: in Queue_Class);
  end Queue_User;

  (...)
```



Concurrent & Distributed Systems



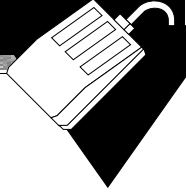
```
task body Queue_Holder is
  Local_Queue : Queue_Class;
  Item        : Element;
begin
  Local_Queue := new Real_Queue; -- could be a different implementation!
  Queue_User.Send_Queue (Local_Queue);
  accept Queue_Filled do
    Dequeue (Item, Local_Queue.all); -- Item will be 'r'
  end Queue_Filled;
end Queue_Holder;

task body Queue_User is
  Local_Queue : Queue_Class;
  Item        : Element;
begin
  Local_Queue := new Real_Queue; -- could be a different implementation!
  accept Send_Queue (Remote_Queue: in Queue_Class) do
    Enqueue ('r', Remote_Queue.all); -- potentially a rpc!
    Enqueue ('l', Local_Queue.all);
  end Send_Queue;
  Queue_Holder.Queue_Filled;
  Dequeue (Item, Local_Queue.all); -- Item will be 'l'
end Queue_User;

begin null; end Queue_Test_Dispatching;
```



Concurrent & Distributed Systems



Ada95

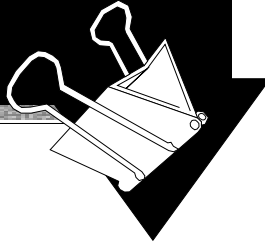
Ada95 language status

- Established language standard with free and commercial compilers available for all major OSs.
- Stand-alone runtime environments for embedded systems (some are only available commercially).
- Special (yet non-standard) extensions (i.e. language reductions and proof systems) for extreme small footprint embedded systems or high integrity real-time environments available ➡ Ravenscar profile systems.
- ➡ has been used and is in use in numberless large scale projects (e.g. in the international space station, and in some spectacular crashes: e.g. Ariane 5)

➡ ***Ada2005 compilers are available now!***



Concurrent & Distributed Systems



Summary

Ada refresher course

- Specification and implementation (body) parts, basic types
- Exceptions
- Information hiding in specifications ('private')
- Generic programming
- Class-wide programming ('tagged types')
- Monitors and synchronisation ('protected', 'entries', 'selects', 'accepts')
 - Abstract types and dispatching