The Ada programming language

An introduction for TTK4145

Kristoffer Nyborg Gregertsen

SINTEF Digital
Department of Mathematics and Cybernetics

2018-02-08

Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary

Contents

Introduction

Basics

- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming

- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

About me

- Department of Mathematics and Cybernetics at SINTEF Digital
- Research manager for automation and real-time systems
  - Group with 8 researchers
  - Implementation and integration of real-time control systems
  - Smart grid and smart infrastructure
  - Industrial automation
  - Robotics
  - Space
- PhD in Engineering Cybernetics from NTNU in 2012
- Real-time support in Ada was the topic of my PhD
- Contributed to ISO standard

Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary

What is Ada?

- Ada is a general purpose programming language designed for safety and maintainability
- High-integrity real-time and embedded systems
- Ordered by the US DoD in the late 70s to replace the hundreds of different languages used in military
- A french team won with the language Ada
- Named after Ada Lovelace
  - The worlds first programmer
  - Therefore **not** acronym ADA
- The language became an ISO standard in 1983

Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary

Ada 2005 and 2012

- Started with Ada 83 and major revision Ada 95
- Ada 2005 improved Ada 95 with features such as:
  - More Java-like object-orientation with interfaces
  - Extensions to the standard library
  - Tasking, real-time improvements
  - The Ravenscar profile for high-integrity systems
- Ada 2012 is the latest revision with features as:
  - Programming by Contract
  - Additional expression and function syntax
  - Task affinities and dispatching domains
  - The Ravenscar profile for multiprocessor systems
- Some minor corrections amended to the standard in 2015

Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary

Why use Ada?

- Ada was designed for use in high-integrity systems and has many safeguards against common programming faults
- Ada has excellent support for development and maintenance of large applications by its notation of packages
- Ada has built-in language support for tasking and a rich set of synchronization primitives
- Ada is used for safety critical projects such as:
  - International Space Station (ISS)
  - Airbus 320, 330, 340, 380
  - Boeing 737, 747-400, 757, 767, 777, 787
  - TGV and European Train Control System (ETCS)
- Ada is a mature language with excellent tool support on many platforms

Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary

Hello world!

```
with Ada.Text_IO;

procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("Hello␣World!");
end Hello_World;
```

- Notice that there are no curly brackets { }
- Ada.Text_IO is a package in the standard library
- Main procedure may have any name
- File called “hello_world.adb”
- Compile with: `gnatmake hello_world`

Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary

Variables

- Naming convention for all identifiers is Like_This
- Ada code is **not** case-sensitive
- Variables are declared with name first, then the type
- Variables may be initialized with a value when declared
- Compiler will warn about use of uninitialized variables

```
procedure My_Procedure is
   I : Integer := 10;
begin
   I := I + 1;
end My_Procedure;
```


Contents
Introduction
Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions
Programming
Routines
Packages
Aspects and contracts
Object-orientation
Standard library
Summary

Constants

- Constants may be of a type or just a named number
- Named numbers are of universal type:
  - No limits in size or precision
  - Somewhat like `#define PI 3.14` in C
- Constants of a type are like variables only ... *constant*

```
Pi      : constant := 3.14159265358979323846264З;
Million : constant := 1_000_000;
Hex     : constant := 16#A5A5_BEEF#;
Binary  : constant := 2#1010010110100101_1011111011101111#;

CI : constant Integer := -1;
```


Contents

Introduction

Basics

Variables and scope

Types and subtypes

Primitive types

Composite types

Basic flow

Exceptions

Programming

Routines

Packages

Aspects and contracts

Object-orientation

Standard library

Summary

Scope

- Variables, types, routines and more all have a scope
- Defined in statement section by **declare – begin – end**
- Also defined by language constructs such as routines

```
procedure My_Procedure is
   X : Float := -1.0;          -- Declarations
begin

   X := X ** 2;                -- Statements, no declarations

   declare
      Y : constant := 0.1;     -- More declarations
   begin
      X := X + Y;              -- Statements
   end;

end My_Procedure;
```


Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary

Types

- Ada is a strongly typed language
- No implicit type-casting as in C
- Two primary classes of types:
  - Primitives
  - Composite
- The primitive types are sub-divided in:
  - Scalars
  - References

Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary

Subtypes

- A type may be defined as a subtype of another
- All values of a subtype are also values of parent
- All values of parent *need* not be values of subtype
- No typecasting is needed from subtype to type

```
declare
   subtype Decimal is Integer range 0 .. 9;
   D : Decimal;
   I : Integer := 1;
begin
   D := Decimal (I);  -- May cause constraint error
   D := D + 1;        -- May cause constraint error
   I := D;            -- Always safe
end;
```


Contents

Introduction

Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions

Programming
Routines
Packages
Aspects and contracts
Object-orientation

Standard library

Summary

Scalars

- Discrete scalars:
  - Enumeration types such as: `Boolean`
  - Integer types such as: `Integer`, `Natural`, `Positive`
- Real scalars:
  - Float types
  - Fixed types
- Range and storage size of types found by:
  - `Integer'First`
  - `Integer'Last`
  - `Integer'Size`

Enumeration

```ada
declare

   type Day is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
   subtype Workday is Day range Monday .. Friday;

   D : Day;
   W : Workday := Monday;

begin
   W := W'Next;               -- After this W is Tuesday
   D := W;                    -- Always safe
   W := Workday (D'Prev);     -- After this W is Monday
end;
```


Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary

Integers

- Ordinary integers and modular numbers
- Integers get constraint error when out of range
- Modular numbers wrap around
- Modular numbers also have binary-operators: **and**, **or**, **xor**

```
declare
   type Word is mod 2 ** 32;
   type Count is range 0 .. 2 ** 32 - 1;

   W : Word  := Word'Last;  -- 2 ** 32 - 1
   C : Count := Count'Last; -- 2 ** 32 - 1
begin
   W := W + 1; -- Will wrap around to 0
   C := C + 1; -- Will cause Constraint_Error!
end;
```


Contents
Introduction
Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions
Programming
Routines
Packages
Aspects and contracts
Object-orientation
Standard library
Summary

Real numbers

- Precision for pre-defined float types is machine dependent
- Possible to define float types with a minimal precision
- Fixed types have a fixed precision and are represented as integers on the hardware, they require no FPU

```
type Real is digits 7;
-- At least 7 digits precision, compiler will chose size

type USD is delta 0.01 range -10.0 ** 15 .. 10.0 ** 15;
-- Fixed precision of 0.01 from -1000 trillion to 1000 trillion
```


Contents

Introduction

Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions

Programming
Routines
Packages
Aspects and contracts
Object-orientation

Standard library

Summary

References

- May define access types for any type and routine
- Three classes of access types:
  - Those that may point only to objects on the heap
  - Those that may point to any object declared as **aliased**
  - Anonymous access types that may point to any object
- Dereferenced using the **all** operator
- Need no explicit dereference when unambiguous
- Heap memory allocated with the **new** operator
- No garbage collector, need explicit deallocation!
- References are less used in Ada than in C

Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary

Example

```
declare

   type Heap_Access is access Integer;
   type All_Access is access all Integer;

   I : aliased Integer := 1;
   A : Heap_Access;                  -- null by default
   B : All_Access      := I'Access;  -- Points to I
   C : access Integer  := B;         -- Points to I

begin
  A      := new Integer'(2);  -- Create integer with value 2
  B.all  := 3;                -- After this I = 3
  B      := A;                -- After this B points to heap
  B.all  := A.all + C.all;    -- After this A.all = B.all = 5
end;
```

Composite

- Composite types may contain:
  - Primitives
  - Other composite types
- Five classes of composite types:
  - **array**
  - **record**
  - **interface**
  - **protected**
  - **task**
- The three latter are discussed later

Array

- Consists of elements of *one* type
- May have several dimensions
- Any discrete type may be index
- May have anonymous array types
- An array type may have fixed or varying length by use of <>

```
type Vector is array (Integer range <>) of Real;

type Matrix is array (Integer range <>, Integer range <>)
  of Real;
```

Example

```
declare
   V : Vector (-10 .. 10) := (0 => 1.0, others => 0.0);

   M : Matrix := ((1.0, 2.0, 3.0),
                  (4.0, 5.0, 6.0),
                  (7.0, 8.0, 9.0));

   A : array (Weekday) of Natural := (Friday => 1,
                                      others => 0);

begin
   V (-10)     := V (0) + 2.0;                -- Assignment
   V (2 .. 3)  := (5.0, 6.0);                 -- Slice assignment
   V           := V (-9 .. 10) & V (-10);     -- Rotate vector
   M (1, 1)    := M (2, 2);                   -- Two dimensional
   A (Monday)  := 2;                          -- Enumeration index
end;
```

Strings

- Three string types are defined in the standard library:
  - String which is an array of Character
  - Bounded_String with varying bounded length
  - Unbounded_String with varying unbounded length
- Length of String is fixed after declaration!
- Use unbounded string to get C++/Java-like strings

```
declare
   A : String := "Hello";
   B : String (1 .. 8);
   C : Unbounded_String := To_Unbounded_String (A);
begin
   B := A & "...";        -- Need same length for assignment
   C := C & " World!";    -- Append string to C
end;
```


Contents
Introduction
Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions
Programming
Routines
Packages
Aspects and contracts
Object-orientation
Standard library
Summary

Record

- Similar to struct in C (but more powerful of course)
- May be defined with default values as shown below

```
declare

   type Complex is
      record
         Re : Float := 0.0;
         Im : Float := 0.0;
      end record;

   A : Complex := (Re => 1.0, Im => 1.0);
   B : Complex := (A.Im, A.Re);

begin
   B.Re := B.Re ** 2;
end;
```


Contents
Introduction
Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions
Programming
Routines
Packages
Aspects and contracts
Object-orientation
Standard library
Summary

Program flow

- Ada supports standard program flow constructs:
  - **if** ... **then** ... **elsif** ... **else**
  - **case**
  - **loop**
  - **for**
  - **while**
  - **goto** (!!!)
- We'll discuss all constructs but **goto**
- The **goto** statement is considered harmful but is needed for automated code generators and in some special cases

if ... then ... elsif ... else

- Uses Boolean expressions (only)
- Boolean expressions need to be grouped with ()
- Notice = for equality and /= for inequality
- The **elsif** keyword removes ambiguity
- The **elsif** and **else** parts are optional

```
if (A and B) or C then
   ...
elsif (X = Y) xor (X /= Z) then
   ...
else
   ...
end if;
```

Case

- Like switch in C, but more powerful and no fall-through
- Works for all discrete types (integer and enumeration types)
- Character used in example below

```
case Input is
   when 'u' =>
      ...
   when 'x' | 'X' | 'q' | 'Q' =>
      ...
   when 'a' .. 'e' =>
      ...
   when others =>
      ...
end case;
```


Contents

Introduction

Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions

Programming
Routines
Packages
Aspects and contracts
Object-orientation

Standard library

Summary

Loop

- Ada has a construct for an eternal loop
- Broken by keyword **exit**
- Loops may be named (removes need for evil **goto**)

```
loop
   ...
   exit when Answer = 43;
   ...
end loop;

Outer:
loop
   ...
   loop
      ...
      exit Outer when Answer = 43;
   end loop;
   ...
end loop Outer;
```


Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary

For

- Iterates over a given discrete range
- The iterator is a constant within loop
- Use keyword **reverse** to iterate in reverse order
- May be broken using **exit**

```
for I in 1 .. 10 loop
   for J in reverse 1 .. 10 loop
      ...
   end loop;
end loop;

for D in Day loop
   ...
end loop;
```


Contents

Introduction

Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions

Programming
Routines
Packages
Aspects and contracts
Object-orientation

Standard library

Summary

While

- Like while in C
- Iterates as long as Boolean expression is true
- May be broken using **exit**

```
declare
   X : Float := 1.00000000001;
begin
   while X < 1000.0 loop
      X := X ** 2;
   end loop;
end;
```


Contents

Introduction

Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions

Programming
Routines
Packages
Aspects and contracts
Object-orientation

Standard library

Summary

Exceptions

- Exceptions are used for error handling
- There are some predefined exceptions:
  - Constraint_Error when value out of range
  - Program_Error for broken program control structure
  - Storage_Error when memory allocation fails
- User defined exceptions are allowed
- Exceptions are handled before **end** in a block
- After an exception is handled the block is left
- Unhandled exceptions propagate downward on call stack, program halts with error message when bottom is reached

Contents

Introduction

Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions

Programming
Routines
Packages
Aspects and contracts
Object-orientation

Standard library

Summary

Example

```
declare
   Wrong_Answer : exception;
begin
   ...
   if Answer /= 43 then
      raise Wrong_Answer with "Answer␣not␣43";
   end if;
   ...
exception
   when Wrong_Answer =>
      Answer := 43;
   when E : others =>
      Put_Line (Exception_Message (E));
end;
```

Routines

- There are two types of routines:
  - Procedures without return value
  - Functions with return value
- Functions *should* not have side effects
- No empty () for routines without arguments
- Routines may have default values for arguments

Procedures

- Arguments of procedures may by marked as:
  - **in**, read only (default)
  - **out**, write only
  - **in out**, read and write
- Arguments marked as **in** may be passed by copy
- Arguments marked as **out** and **in out** passed by reference
- Two procedure specifications (prototypes) are shown below

```
procedure Swap (A, B : in out Integer);
procedure Print (S : String; N : Natural := 1);
```

Example

```
procedure Swap (A, B : in out Integer) is
   T : Integer := A;
begin
   A := B;
   B := T;
end Swap;

procedure Print (S : String; N : Natural := 1) is
begin
   for I in 1 .. N loop
      Put_Line (S);
   end loop;
end Print;
...
Swap (This, That);             -- Ordinary call
Swap (B => That, A => This);  -- Named arguments
Print ("Hello", 10);
Print ("World!");
```

Functions

- Defaults to **in** but Ada 2012 also allows **in out**
- Ada 2012 adds expression function syntax

```
function Sum (A, B : Integer) return Integer is
begin
   return A + B;
end Sum;

function Product (A, B : Integer) return Integer is (A * B);
...
declare
   C : Integer;
begin
   C := Sum (1, 2);
   C := Product (C, 3);
end;
```

Overloading

- Several routines may have the same name
- Which routine is called depends on:
  - Argument types for procedures and functions
  - Return type for functions
- Overloading decided at compile time
- Needs to be unambiguous, or compilation will fail

Contents
Introduction
Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions
Programming
Routines
Packages
Aspects and contracts
Object-orientation
Standard library
Summary

Packages

- The building blocks of Ada applications
- Two parts the specification (.ads) and body (.adb):
- Specification has a public and a private section:
  - Public section contain declarations visible to users
  - Private section allows to hide complexity – **abstraction**
  - Public section may define a limited view of types
  - Private section defines the full type – **Abstract Data Types**
- Body contains implementation of routines
- The body may also have internal declarations and routines

Contents

Introduction

Basics

Variables and scope

Types and subtypes

Primitive types

Composite types

Basic flow

Exceptions

Programming

Routines

Packages

Aspects and contracts

Object-orientation

Standard library

Summary

Example

```ada
-- File: simple_queue.ads

package Simple_Queue is

   type Queue is limited private;

   procedure Enqueue (Q : in out Queue;
                      E : in     Item);

   procedure Dequeue (Q : in out Queue;
                      E :    out Item);

   function Length (Q : Queue) return Natural;

private
   type Queue is
      record
         ...
      end record;
end Simple_Queue;
```

Example

```
-- File: simple_queue.adb

package body Simple_Queue is
   procedure Enqueue (Q : in out Queue;
                      E : in     Item) is
   begin
      ...
   end Enqueue;

   procedure Dequeue (Q : in out Queue;
                      E :    out Item) is
   begin
      ...
   end Dequeue;

   function Length (Q : Queue) return Natural is
   begin
      ...
   end Length;
end Simple_Queue;
```

Example

```
-- File: test.adb

with Simple_Queue;
use Simple_Queue;

procedure Test is
   A, B, I : Item;
   Q : Queue;
begin
   ...
   Enqueue (Q, A);
   Enqueue (Q, B);
   ...
   while Length (Q) > 0 loop
      Dequeue (Q, I);
      ...
   end if;
end Test;
```

Programming by contract

- Ada 2012 adds the notion of *aspects*:
  - Partly same functionality as pragmas
  - Specify compile attributes of data and routines
- Programming by contract is one use of aspects:
  - Routine pre- and post-conditions
  - Data type invariants
- Checked at run-time same as assertions
- Used in SPARK for formal verification of code properties

Contents
Introduction
Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions
Programming
Routines
Packages
Aspects and contracts
Object-orientation
Standard library
Summary

Example

```ada
package Simple_Queue is

   Max_Length : constant := ...

   type Queue is limited private;

   procedure Enqueue (Q : in out Queue;
                      E : in     Item)
     with
       Pre => not Is_Full(Q),
       Post => (Length (Q) = Length (Q)'Old + 1);

   procedure Dequeue (Q : in out Queue;
                      E :    out Item)
     with
       Pre => not Is_Empty(Q),
       Post => (Length (Q) = Length (Q)'Old - 1);

   function Length (Q : Queue) return Natural;
   function Is_Full (Q : Queue) return Boolean is (Length (Q) = Max_Length);
   function Is_Empty (Q : Queue) return Boolean is (Length (Q) = 0);
...
end Simple_Queue;
```

Example

```
procedure Odd_Even is

   type Even is new Natural
      with Type_Invariant => (Even mod 2 = 0);

   type Odd is new Natural
      with Type_Invariant => (Odd mod 2 = 1);

   E : Even := 0;
   O : Odd  := 1;

begin

   E := E + 2; -- OK
   O := O + 2; -- OK
   E := E + 1; -- Exception

end Odd_Even;
```

Object-orientation

- Similar OO-model as Java:
  - Classes
  - Interfaces
- OO-model based on **tagged** records
- Interfaces were introduced with Ada 2005
- The definition of a class and its methods are usually gathered in a package, no link between class and file as in Java
- Abstract classes may have abstract and null methods
- For interfaces only abstract and null methods are allowed
- Dispatching calls only for class-wide types (Type'Class)

Contents

Introduction

Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions

Programming
Routines
Packages
Aspects and contracts
Object-orientation

Standard library

Summary

Example

```
-- File: shapes.ads

package Shapes is

   -- Abstract base type with no data.
   type Shape is abstract tagged null record;

   -- Abstract procedure must be overloaded.
   procedure Draw (This : Shape) is abstract;

   -- Access any type extending Shape.
   type Any_Shape is access all Shape'Class;

end Shapes;
```

Example

```
-- File: shapes-surfaces.ads

package Shapes.Surfaces is

   -- Interface type has no data.
   type Surface is interface;

   -- Abstract function must be overloaded.
   function Area (This : Surface) return Float is abstract;

   -- Access any type implementing Surface.
   type Any_Surface is access all Surface'Class;

end Shapes.Surfaces;
```

Example

```
-- File: shapes-rectangles.ads

with Shapes.Surfaces;
use Shapes.Surfaces;

package Shapes.Rectangles is

   -- Extends Shape and implements Surface, public data.
   type Rectangle is new Shape and Surface with
      record
         Width, Height : Float;
      end record;

   -- Notice optional overloading keyword.
   overloading procedure Draw (This : Rectangle);
   overloading function Area  (This : Rectangle) return Float;

end Shapes.Rectangles;
```


Contents
Introduction
Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions
Programming
Routines
Packages
Aspects and contracts
Object-orientation
Standard library
Summary

Example

```
-- File: shapes-rectangles.adb

with Ada.Text_IO;
use Ada.Text_IO;

package body Shapes.Rectangles is

   procedure Draw (This : Rectangle) is
   begin
      for I in range 1 .. Integer (This.Height) loop
         for J in range 1 .. Integer (This.Width) loop
            Put ('#');
         end loop;
         New_Line;
      end loop;
   end Draw;

   function Area (This : Rectangle) return Float is (This.Width * This.Height);

end Shapes.Rectangles;
```

Example

```
-- File: test.adb

with Shapes.Surfaces, Shapes.Rectangles;
with Ada.Text_IO, Ada.Float_Text_IO;
use Shapes, Shapes.Surfaces, Shapes.Rectangles;
use Ada.Text_IO, Ada.Float_Text_IO;

procedure Test is
   R : aliased Rectangle := (Height => 1.0, Width => 2.0);
   A : Any_Shape    := Any_Shape (R'Access);
   B : Any_Surface  := Any_Surface (A);
begin
   A.Draw;
   Put (B.Area);
   New_Line;
end Test;
```


Contents
Introduction
Basics
Variables and scope
Types and subtypes
Primitive types
Composite types
Basic flow
Exceptions
Programming
Routines
Packages
Aspects and contracts
Object-orientation
Standard library
Summary

Standard library

- Input / Output
  - Standard input/output
  - Streams and file input/output
- Containers (like C++/STL and Java)
- Real-time features
- Distributed programming
- Linear algebra (built upon LAPACK and BLAS)
- Networking sockets (TCP and UDP)

Summary

- Ada is a programming language most used in safety-critical domains
  - Strong typed and many compiler checks
  - Large systems with packages and abstraction
  - Built-in concurrency and real-time support
- Mature language that has been ISO standard since early 80's
- Latest revision is Ada 2012 with update in 2015
- Excellent tools for a wide range of embedded platforms

Contents

Introduction

Basics

Variables and scope

Types and subtypes

Primitive types

Composite types

Basic flow

Exceptions

Programming

Routines

Packages

Aspects and contracts

Object-orientation

Standard library

Summary

Lecture next week

- Concurrent constructs
- Real-time system support
- Embedded system support
- SPARK for formal verification

Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary

Thank you!

Contents

Introduction

Basics
- Variables and scope
- Types and subtypes
- Primitive types
- Composite types
- Basic flow
- Exceptions

Programming
- Routines
- Packages
- Aspects and contracts
- Object-orientation

Standard library

Summary