Ada: A brief History

- 1975-1979: strawman-steelman requirements, Pascal is base
  - Packages, generics, tasks, derived types
- 1983: ANSI Standard for Ada83, first validated compiler
- 1990-1994: Revision Process
  - Objects, dynamic dispatching, polymorphism, protected types
- 1995: First validated GNAT compiler
- 1995-2004: Revision Process
- 2005 (Expected) new ISO Ada05 Standard
  - Interfaces for multiple inheritance, mutually-recursive types
Design principles

- **Software Engineering**: programming in the large
- Readability over writability
- Rigorous standard for portability
- Precise semantics
- Built-in concurrency
- **Strong-Typing**: all unsafe code is explicit
Ada, C++ and Java

- Big languages win
- strong typing: Ada, C++, Java
- Generics: Ada $\Rightarrow$ C++ $\Rightarrow$ Java 1.5
- Concurrency: Ada $\Rightarrow$ Java
- Packages: Ada $\Rightarrow$ C++ and Java
  - Ahead of its time in design
  - Out of step in implementation
  - Between language generations
Key Goals

- Readability: between COBOL and APL
- strong typing: compile, don’t debug
- Programming in the large: package it
- Exception Handling: Murphy is always right
- Data abstraction: apples are not oranges
- Object orientation: Dispatch and inherit
- Tasking: walk, chew gum, etc.
- Generic units: the easiest reuse is to copy
- Interfacing: play well with others
The canonical example

with Text_io; use Text_io;
procedure example is
begin
  Put_Line ("easier than we thought!");
end;
A small package

Package Math_Functions is

    function Sqrt (X : Float) return Float;

    function Exp (Base : Float; Exponent : Float) return Float;

end Math_Functions;
Using the package

```ada
with Math_Functions; with Gnat_IO; use Gnat_IO;
procedure Example2 is
    Val : Float;
begin
    get (Val);
    put ("Sqrt ("; put (Val); put (" = ");
    put (Math_Functions.Sqrt (Val));
    new_line;
end;
```
Implementing the package

Package body Math_Functions is

  Epsilon : constant := 1.0e-7;

  function Sqrt (X : Float) return Float is
    Result : Float := X / 2.0;
    begin
      while abs (Result * Result - X) > Epsilon loop
        Result := 0.5 * (X / Result + Result);
      end loop;
      return Result;
    end Sqrt;

  end Math_Functions;
Abstraction at its best: enumeration types

- Trivial Implementation: literals are mapped into successive integers
- Very common abstraction: list of names, properties
- Expressive of real-world domain, hides machine representation
  - type suit is (hearts, diamonds, spades, clubs); 
  - type direction is (east, west, north, south, lost);

- Order of list implies that spades > hearts, etc.
- Contrast (C#): “arithmetics on enum members may produce results in the underlying representation type that do not correspond to any declared enum member; this is not an error”
Enumeration types and strong typing

```ada
type Fruit is (Apple, Orange, Grape, Apricot);
type Vendor is (Apple, IBM, HP, Lenovo);
My_PC : Vendor;
Dessert : Fruit;
...
My_PC := Apple;
Dessert := Apple;
Dessert := My_PC;                -- ERROR
```
Built-in enumeration types

type Boolean is (False, True);

type Character is (....) -- full ASCII (Ada83) or Latin_1 (Ada95)
    -- not expressible in Ada

type Wide_Character is (....) -- Unicode, or ISO646
Array Types

Index type(s) of an array type are typed:

```ada
type weekday is (Mon, Tue, Wed, Thu, Fri);

type workhours is array (Weekday) of integer;

type pressure is array (1..1000, 1..1000, 1..1000) of Long_Float;
```

Predefined array:

```ada
type String is array (Positive range <>) of Character;
```
Record Types

c conventional named fields:

type Buffer is record
    size : Positive;
    contents : String (1 .. 100);
end record;

B1 : Buffer; -- can use B1, B1.size, B1.contents (10)
Access Types

Typed pointers, for type safety and to minimize aliasing:

```
type Handle is access Buffer;

Ptr : Handle := new Buffer;

Ptr.all is a Buffer.
Can write Ptr.size, Ptr.contents, etc.
```
Abstraction mechanisms

- Packages
- Private types
- Objects and Inheritance
- Classes, polymorphism, dynamic dispatching
- Generic units
- Concurrency: tasks and protected types
Packages

- A related set of types, constants, and subprograms
- Separate declaration (interface) and implementation
- Supports privacy and data hiding
- The single most important idea in software engineering
A package for stacks

package Stacks is
  type Stack is private;
  procedure Push (It : Character; On : in out Stack);
  procedure Pop   (It : Character; From : in out Stack);
  function Empty (S : Stack) return Boolean;

private
  type Stack is record
    top : Integer := 0;
    contents : String (1 .. 80) := (others => '*');
  end record;
end Stacks;
Object-oriented Programming

- Type extension
- Inheritance and overriding
- Run-time dispatching
- Polymorphism
- Class-wide programming
- Abstract types and subprograms
Type Extension

- Mechanism to define new types by enrichment:

```ada
type Point is tagged record
    X_Coord, Y_Coord : Integer;
end record;

type Pixel is new Point with record
    R, G, B : Integer;
end record;
```
Inheritance

- A type has primitive operations: operations that involve the type as a parameter or a returned value.
- A type extension inherits the primitive operations of its parent.
- A primitive operation can be redefined and overridden for a descendant type.
Polymorphism

- A class is a family of types with the same ancestor.
- An object of the class is identified at run-time by its tag.
- Dynamic dispatching uses the tag of the object to determine the operation that applies.
- A classwide operation applies uniformly to all member of the class:

  procedure Examine (Thing : in out T’Class);
Generic Units

- The basic tool for software reuse.
- Parameters can be types, objects, subprograms, packages.
- Akin to C++ templates.
- Absence from Java is incomprehensible
A Generic Package

Generic
  type T is private;
package Stacks is
  type Stack is private;
  procedure Push (Thing : T ; On : in out Stack);
  ...
private
  type Arr is array (1 .. 100) of T;
  type stack is record
    Top : Integer := 100;
    contents : Arr;
  end record;
end Stacks;
A Generic Subprogram

Generic
    type T is private;
    type Arr is array (Integer range <>) of T;
    with function "<" (X, Y : T) return Boolean;
procedure Sort (Table : in out Arr);
The Type Model

- Types and Subtypes
- Declarations and their scope
- Objects, constants and variables
- Scalar types
- Array types
- Record types
- Access types
Types and Subtypes

- A type is a set of values and a group of operations
- A type declaration specifies static properties of objects
- A subtype declaration specifies dynamic properties of values.
- Types with different names are distinct and incompatible: name equivalence everywhere, instead of structural equivalence.
- Subtypes of the same base type are compatible.
Compile-time vs. run-time

- Type properties are enforced by the compiler:
  - `x : integer := false;`
    - program rejected
  - `x : positive := f (z);`
    - if f returns an integer, need to check value at run-time
Built-in subtypes

```ada
type Integer is .. -- implementation defined
subtype Positive is integer range 1 .. Integer’Last; -- useful attribute
subtype Natural is integer range 0 .. Integer’Last;
X : integer := 500;
Y : Positive := 2 * X;
Z : Natural := - Y; -- legal, raises constraint error
```
Declarations and Scope

- Declarations are elaborated in order
- Entities become visible at the end of their declaration (usually)
- Block structure allows arbitrary nesting of declarative regions.
- Declarations can appear in
  - subprograms
  - packages
  - blocks
  - ...
Blocks

Declare

X : Integer := F (5);
Y : Integer := 2 * X;
Z : Integer := Y * Z;               -- Error: premature
X : Float ;                          -- Error: duplicate
begin
  declare
    X : Float := Float (Y);          -- hides outer X
  begin
    Put_Line (Float'Image (X));
  end;
end;
Variables and Constants

Variable declaration:

Limit : Integer := 25;
Offset : Integer range 1 .. 20;

Constant declaration:

Sqrt2  : constant float := Sqrt (2.0);  -- not static
Always : constant Boolean := True;     -- static value
Never  : constant Boolean := not Always; -- static expression
Variables must be constrained

Subtype is constrained:

First_Name : String (1..5) := “Ralph”;

but not necessarily static:

Last_Name : String (1 .. X * 2);

else subtype is indefinite but initial value provides bounds:

Comment : String := “this is obvious”; -- bounds are 1 .. 15
Multiple Declarations

This, That : T := F (1, 2, 3);

Is equivalent to

This : T := F (1, 2, 3);
That : T := F (1, 2, 3);

F is called twice. Important if expression has side-effect:

type  Ptr  is access R;
P1, P2 : Ptr := new R;

two R’s are allocated.
Number Declarations

\[
\begin{align*}
\Pi & : \text{constant} := 3.14159265; & \quad & \text{-- type deduced from value} \\
\text{Half Pi} & : \text{constant} := \Pi / 2; & \quad & \text{-- mixed arithmetic OK} \\
\text{Big} & : \text{constant} := 2 \times 200; & \quad & \text{-- legal} \\
\text{One} & : \text{constant} := 2 \times \text{Big} / (\text{Big} + \text{Big}); & \quad & \text{-- must be exact}
\end{align*}
\]
Scalar Types

- Discrete types
  - Integer types
  - Enumeration Types
- Real types
  - Floating-point types
  - Fixed_point types
Integer Types

Several predefined types: Integer, Long_Integer, Short_Integer

Specific bounds of type must be static:

```plaintext
type My_Int is range -2 ** 16 .. 2 ** 16 - 1;
type Tiny is range 0 .. 10;
```

By giving explicit bounds, program is more portable: each compiler will figure out what hardware type corresponds

Modular types:

```plaintext
type Byte is mod 2 ** 8;
```
Integer Operations

Comparison Operators:          =  /=  <  <=  >  >=
Addition Operators:            +  -
Unary operators               +  -
Multiplying operators         *  /  mod  rem
Highest precedence Operators: **  abs
Boolean Operations

All attributes of discrete types

Boolean binary operators : and or xor
Boolean unary operators : not
Short-circuit operations : and then or else
Membership operations : in not in

When in doubt, parenthesize!
Attributes

- Attributes denote properties of a type, or type-specific properties of a value
  - Boolean’Size -- 1, because single bit
  - character’size -- 8 bits
  - month’pos (Jul) -- involves type and literal
  - table’length (1) -- specify array and dimension

- Could be written as a function, but functions don’t take types as arguments => need new syntax.
Attributes on Discrete Types

Byte’First, Long_Integer’Last -- applies to type or subtype
Weekday’Succ (Today) -- like function call
Integer’Succ (X*Y) -- like adding one
Boolean’Pred (True) -- Yields False
Boolean’Succ (True) -- Exception
Weekday’Pos (Mon) -- Yields 0
Weekday’Val (3) -- Yields Thu
Positive’Max (X, Y) -- function with two args
Real Types

All computations are approximate:

Fixed point type: absolute bound on error:

```
type temp is delta 2 ** (-16) range -100.0 .. 100.0;
```

Floating point type: relative bound on error:

```
type Angle is digits 7 range -2.0 .. 2.0;
```

Predefined floating point types: Float, Long_Float, etc.
Derived Types

A general mechanism for creating new types with the properties of existing ones:

type Like_T is new T; -- same set of values, same operations.

type Small_Int is range 1 .. 10;

equivalent to

type Anon is new Integer;

subtype Small_Int is Anon range 1 .. 10;

and all arithmetic operations are inherited
Array Types

Index types can be of any discrete type
Component type must be definite:

\[
\text{type} \text{ class\_list is array (1 .. 100) of String (1..10);} \quad \text{-- OK}
\]
\[
\text{type} \text{ class\_list is array (1 .. 100) of String;} \quad \text{-- Error}
\]

Subtype constrains all indices or none:

\[
\text{type} \text{ Matrix is array}
(\text{positive range <>}, \text{positive range <>}) \text{ of Long\_Float;}
\]
\[
\text{subtype} \text{ Table is Matrix;}
\]
\[
\text{subtype} \text{ Rotation is Matrix (1 .. 3, 1 .. 3);} 
\]
Anonymous Array Types

Grades : array (1 .. Num_Students) of Natural;

type of Grades has no name: distinct from any other array types.

Ar1, Ar2 : array (1 .. 10) of Boolean;
...

Ar1 := Ar2;    -- Error: different (anonymous) types.

If type is useful, it deserves to have a name.
type Matrix is array (Positive range <> , Positive range <> )
          of Float;
subtype Rect is Matrix (1 .. 3, 1 .. 5);
M3 : Rect;
  M3’First (1)         --  Yields 1
  M3’First            --  same.
  Rect’length (2)     --  Yields 5 (applies to type)
  M3’range (2)        --  equivalent to 1..5
  String’Length       --  ERROR: unconstrained
Array Aggregates

Expression that yields an array value:

- \( A := (1, 2, 3, 10); \) \hspace{1cm} -- \text{positional}
- \( A := (1, \text{others} \Rightarrow 0); \) \hspace{1cm} -- \text{notation for default.}
- \( A := (1..3 \Rightarrow 1, 4 \Rightarrow -999); \) \hspace{1cm} -- \text{component associations}

Default can only be used if bounds are known:

- \( A : \text{String} (1 .. 10) := (\text{others} \Rightarrow '?'); \) \hspace{1cm} -- OK
- \( A : \text{String} := (\text{others} \Rightarrow '?'); \) \hspace{1cm} -- Error: unknown bounds.
Aggregates and Qualification

Aggregate may be ambiguous:

```ada
    type Vector is array (1 .. 3) of Float;
    procedure Display (V : vector);

    type Assay is array (1 .. 3) of Float;
    procedure Display (A : assay);

    Display ((1.0, 1.2, 1.5)); -- ambiguous
    Display (Vector ' (1.0, 1.2, 1.5)); -- OK.
```
Multidimensional Arrays

Aggregates given in row-major order with subaggregates:

```ada
    type Square is array (1 .. 3, 1 .. 3) of Integer;
    Unit : constant Square := ( (1, 0 ,0), (0, 1, 0), (0, 0, 1));
```

Two-dimensional array is NOT array of arrays:

```ada
    type vector is array (1 .. 3) of Integer;
    type V3 is array (1 .. 3) of vector;
    -- not convertible to Square
```
Boolean operations extend pointwise:

```ada
  type Set is array (1 .. Card) of Boolean;
  S1, S2, S3 : Set;
  ...
  S3 := S1 and S2; -- Intersection
```

Lexicographic comparisons on arrays of discrete types:

```ada
  S1 := (T, T, T);
  S2 := (T, T, F);
  .. S2 < S1 -- yields True
```
Concatenation and Slicing

Both operations yield the base type:

```ada
    type Table is array (1..10) of Integer;
    T1, T2 : Table;
...
    T1 & T2   -- What type?
```

Declaration equivalent to:

```ada
    type Anon is array (integer range <>) of Integer;
    subtype Table is Anon (1 .. 10);
    T1 & T2 , T1 (X .. Y) are of type Anon
```
Specifying a range

subtype Sub is Positive range 2 .. 4;
Label : String (1..10) := “transcends” ;

... 

Label (2 .. 4)                        --  Yields “ran”
Label (Integer range 2 .. 4)          --  Same
Label (Sub)                           --  Ditto

Also used in loops and case statements.
Control Structures

Conventional sequential constructs:

- If-Statement
- Loop
- Case statement
- Goto and labels

More novel forms for task communication.
If Done (X, Y) then
  Success;
  Close_Up;
elsif Almost_Done (X) then  -- the only keyword that isn't English
  Hurry (Y);
else
  if X = 0 then Call_For_Help (X) else Panic; end if;
end if;
Loops

Infinite loop:

```
loop
    Put_Line ("Forever");
end loop;
```

In general, better to stop:

```
loop
    Put_Line ("Still_Going");
    exit when Time_Is_Up;        -- must be boolean value
end loop;
```
Loops over discrete ranges

for J in 1 .. 1000 loop  -- declares J

for K in 1 .. 0 loop  -- empty range

for Month in Feb .. Nov loop
for Month in Months range Feb .. Nov loop

for K in Positive loop  -- might take a long time

for Num in reverse 1 .. 1000 loop  -- descending order
While-Loops

while abs (Result * Result - X ) > Epsilon loop
Result := 0.5 * (X / Result + Result);
end loop;

Effect of Until can be obtained with while and exit statements.
Named Loops

search: \textbf{while} \( X > 0 \) \textbf{loop}
\[X := F(X, Y);\]

\textbf{refine:} \textbf{for} \( J \) in 1 .. N \textbf{loop}
\[Y := G(Z);\]
\textbf{exit search when} \( X = 0.0;\)
\textbf{end loop refine;}

\textbf{if} \( T > 20 \) \textbf{then exit; end if;} \hspace{1cm} \text{-- alternate form}
\textbf{end loop search;}

.
Case Statements

Most programs are interpreters for some abstract machine

=> case statement is most useful control structure!

Works naturally with discrete types, in particular enumerations.

Case alternatives must cover all values of the range
Case Statements

X : Integer;
..
Case (X+1) is -- Expression of integer type
  when Integer’First .. 0 => Handle_Non_Positive;
  when 2 .. 4 | 6 | 8 => Process (X);
  when 9 .. 12 | 2 => null;
  when others => Use_Default (X); -- Required
    Display (X);
end case;

can use qualification to restrict range: case Int’(x+1) is ...
Goto Statement

Occasionally useful.
Syntax of maximum ugliness to discourage use:

```
while Going loop
  Compute_Some;
  if Hopeless then goto next_try; end if;
  Compute_Some=More;
  <<next_try>> Adjust_Computation;
end loop;
```

Restricted range. Raise statement is wild jump.
Subprograms

- Functions and procedures.
- Functions only have in-parameters.
- Functions may return unconstrained types.
- Function execution must end in return statement.
- Parameter passing by copy-return for scalar types.
- Parameter passing not specified for non-tagged types.
- Positional and named notation in calls.
- Defaults for in-parameters.
function F (X : integer := 0;
    Y : Float := Pi;
    Maybe : Boolean := True)
return Integer;

...

F (10, 0.0, False)
F (5)  -- equivalent to F (5, Pi, True)
F (Maybe => False)  -- equivalent to F(0, Pi, False)
Operators

Like functions, but usable in infix notation.

Package Bignums is
  type Bignum is private;
  Zero : constant Bignum; -- deferred constant
  function "+" (X, Y : Bignum) return Bignum;
  function "*" (X, Y : Bignum) return Bignum;
  function Image (X : Bignum) return String;
private ..
End Bignums;

Must respect syntax: no defaults, no unary "*", etc.