Ada: Eine Einführung

Jonathan Dees

Universität Karlsruhe

Seminar Formale Software Entwicklung SS 2007

Gliederung

Ziele des Vortrags

- Ada Code verstehen können
- Fähigkeiten der Sprache kennen lernen
- *Gefühl* für die Sprache entwickeln

Was ist Ada?

- Programmiersprache
- 1970 vom US DoD entwickelt
- Namensgeberin Lady Ada Lovelace

Wozu neue Programmiersprache?

- jede Programmiersprache hat seine Vor- und Nachteile
- das Problem entscheidet die Programmiersprache
- es gibt keine *beste* Programmiersprache

Geschichte (1)

- 1970 mehr als 450 (teils proprietäre) Programmiersprachen beim US DoD
- ⊖ Wartung, Ausbildung, Modularität, Wiederverwendung

Geschichte (1)

- 1970 mehr als 450 (teils proprietäre) Programmiersprachen beim US DoD
- ⊖ Wartung, Ausbildung, Modularität, Wiederverwendung
- Bildung einer Arbeitsgruppe zur Anforderungserhebung und Standardisierung

Geschichte (1)

- 1970 mehr als 450 (teils proprietäre) Programmiersprachen beim US DoD
- $\ominus$ Wartung, Ausbildung, Modularität, Wiederverwendung
- Bildung einer Arbeitsgruppe zur Anforderungserhebung und Standardisierung
- keine bestehende Programmiersprache erfüllte Anforderungen

$\Rightarrow$ Geburt von Ada

Geschichte (2)

- Standard Ada 1815 (ANSI/ISO) (1980)
- Auflage DoD: jedes Softwareprojekt Ada Code > 30% (bis 1997)

Geschichte (2)

- Standard Ada 1815 (ANSI/ISO) (1980)
- Auflage DoD: jedes Softwareprojekt Ada Code > 30% (bis 1997)
- Standard Ada 1995 (ANSI/ISO)
- Objektorientiert (erste genormte OO Sprache)

Geschichte (2)

- Standard Ada 1815 (ANSI/ISO) (1980)
- Auflage DoD: jedes Softwareprojekt Ada Code > 30% (bis 1997)
- Standard Ada 1995 (ANSI/ISO)
- Objektorientiert (erste genormte OO Sprache)
- freier! Standard Ada 2005 (ANSI/ISO) (seit Januar 2007)
- US Air Force finanziert freien GNAT-Compiler

Ada Eigenschaften (Ada2005)

- streng typisiert
- Objektorientiert (mit dynamischer Polymorphie)
- Echtzeitunterstützung
- optionale automatische Speicherbereinigung
- Ausnahmebehandlung
- parallele Verarbeitung
- Schnittstellen zu anderen Sprachen
- (optionale) Tests zur Laufzeit
- unterstützt Prüfung auf formale Korrektheit

Einsatz

hauptsächlich sicherheitskritische Bereiche mit Echtzeitbezug

- Flugsicherung
- Sicherungssystem Eisenbahn
- Waffensystemen
- Raumfahrt (fatales Beispiel: Ariane 5)
- Medizin
- Kernkraftwerke

Hello World

```
with Ada.Text_IO;

procedure Hello is
begin
   Ada.Text_IO.Put_Line("I'm Living!");
end Hello;
```

Programmeinheiten

Logische Einheiten

- subprogram
- package
- spezielle unit (task unit, protected unit, generic unit)

Programmeinheiten

Logische Einheiten

- subprogram
- package
- spezielle unit (task unit, protected unit, generic unit)

Je Teilung in zwei Dateien

- Spezifikation (*.ads*)
- Implementation (*.adb*)

Namenskonventionen

keine Unterscheidung zwischen Groß- und Kleinschreibung
Längenunterstützung per Compiler mindestens 200 Zeichen

Namenskonventionen

keine Unterscheidung zwischen Groß- und Kleinschreibung
Längenunterstützung per Compiler mindestens 200 Zeichen

Bezeichner Allgemein

- Grossbuchstaben_Am_Anfang
- Unterstriche_Zwischen_Wörtern
- Namen, die die Entität am besten beschreiben
- kurze Namen
- kurze Synonyme anstelle von Abkürzungen

Namenskonventionen

keine Unterscheidung zwischen Groß- und Kleinschreibung
Längenunterstützung per Compiler mindestens 200 Zeichen

Bezeichner Allgemein

- Grossbuchstaben_Am_Anfang
- Unterstriche_Zwischen_Wörtern
- Namen, die die Entität am besten beschreiben
- kurze Namen
- kurze Synonyme anstelle von Abkürzungen

Vordefinierte Bezeichern

- durchweg Kleinbuchstaben

Namenskonventionen Beispiele

Unkonventionell

max_Integer maxInteger
UK Universitaet_Karlsruhe
TOD
Left_Leaf_Of_Tree
With Ada.Text_IO;
Elements_Are_Down_To_Zero

Empfohlen

Namenskonventionen Beispiele

Unkonventionell

max_Integer maxInteger
UK Universitaet_Karlsruhe
TOD
Left_Leaf_Of_Tree
With Ada.Text_IO;
Elements_Are_Down_To_Zero

Empfohlen

Max_Integer
Uni_Karlsruhe
Time_Of_Day
Left
with Ada.Text_IO;
Is_Empty

Basisdatentypen

- Integer
- Float
- Character
- String
- Boolean

Integer

Ganzahl (mindestens 16 Bit)

- Arithmetik: $-$ + $*$ / **mod rem** $**$
- Vergleiche: = =/ < > <= >=
- Wertebereich: Integer'First bis Integer'Last

Integer

Ganzahl (mindestens 16 Bit)

- Arithmetik: $-$ + $*$ / **mod rem** $**$
- Vergleiche: = =/ < > <= >=
- Wertebereich: Integer'First bis Integer'Last

Code

```
Put("Integer from "); Put(Integer'First);
Put(" to"); Put(Integer'Last);
```

Integer

Ganzahl (mindestens 16 Bit)

- Arithmetik: $-$ + $*$ / **mod rem** $**$
- Vergleiche: = =/ < > <= >=
- Wertebereich: Integer'First bis Integer'Last

Code

```
Put("Integer from "); Put(Integer'First);
Put(" to"); Put(Integer'Last);
```

Ausgabe

```
Integer from -2147483648 to 2147483647
```

Float

Fließkommazahl (mindestens 16 Bit)

- Arithmetik: $-$ + $*$ / **mod rem** $**$
- Vergleiche: = =/ < > <= >=
- Minimum: Float'First
- Maximum: Float'Last
- Anzahl signifikante Stellen: Float'Digits
- kleinste positive Zahl: Float'Small

Zahlenliterale

```
numeric_literal ::= decimal_literal | based_literal
decimal_literal ::= numeral [ . numeral ] [ exponent ]
numeral ::= digit { digit | "_" }
exponent ::= " E" [ "+" | " ] numeral
based_literal ::= base "#" based_numeral "#" [ exponent ]
base ::= numeral
based_numeral ::= extended_digit { extended_digit | "_" }
extended_digit ::= digit | " A" | "B" | C" | "D" | " E" | " F"
```

Zahlenliterale

```
numeric_literal ::= decimal_literal | based_literal
decimal_literal ::= numeral [ . numeral ] [ exponent ]
numeral ::= digit { digit | "_" }
exponent ::= " E" [ "+" | " ] numeral
based_literal ::= base "#" based_numeral "#" [ exponent ]
base ::= numeral
based_numeral ::= extended_digit { extended_digit | "_" }
extended_digit ::= digit | " A" | "B" | C" | "D" | " E" | " F"
```

Zahlenliterale

```
numeric_literal ::= decimal_literal | based_literal
decimal_literal ::= numeral [ . numeral ] [ exponent ]
numeral ::= digit { digit | "_" }
exponent ::= " E" [ "+" | " ] numeral
based_literal ::= base "#" based_numeral "#" [ exponent ]
base ::= numeral
based_numeral ::= extended_digit { extended_digit | "_" }
extended_digit ::= digit | " A" | "B" | C" | "D" | " E" | " F"
```

Zahlenliterale Beispiel

Zahlenbeispiele

2_000_000_000
2#1001_1000#
3#0.1#
314.159E−2
16#F00#

Unterscheidung zwischen Int und Float anhand des Punktes

Character

Einfaches Zeichen (8 Bit)
Darstellung: ’A’ ’ ’ ’ ’"’

- ASCII Code: Character’Pos(’A’) = 65
- Zeichen über Code: Character’Val(96) = ’a’

Keine Escape Zeichen wie in C

String

Array von Charactern, *nicht* Null-Terminiert
Darstellung: "Sie sagte ""Wir werden sehen!""."

- Konkatenation: &
- Zugriff auf einzelne Elemente: My_String(10);
- Vergleiche lexikographisch: = /= < > <= >=

Arbeitsweise wie mit Arrays

Boolean

Wahrheitswert ausschließlich true oder false
$\neq 0$ bzw $\neq 1$

- Operatoren: **and or not xor**
- spezielle Kurzauswertung: **and then** bzw **or else**

Boolean

Wahrheitswert ausschließlich true oder false
$\neq 0$ bzw $\neq 1$

- Operatoren: **and or not xor**
- spezielle Kurzauswertung: **and then** bzw **or else**

Beispiel

if N=/0 **and** 10 / N = 1 **then** . . .

Boolean

Wahrheitswert ausschließlich true oder false
$\neq 0$ bzw $\neq 1$

- Operatoren: **and or not xor**
- spezielle Kurzauswertung: **and then** bzw **or else**

Beispiel

```
if N=/0 and then 10 / N = 1 then ...
```

Auswertungsreihenfolge unbestimmt, kein frühzeitiger Abbruch

Boolean

Wahrheitswert ausschließlich true oder false
$\neq 0$ bzw $\neq 1$

- Operatoren: **and or not xor**
- spezielle Kurzauswertung: **and then** bzw **or else**

Beispiel

if A **and** B **or** C **then** ...

Boolean

Wahrheitswert ausschließlich true oder false
$\neq 0$ bzw $\neq 1$

- Operatoren: **and or not xor**
- spezielle Kurzauswertung: **and then** bzw **or else**

Beispiel

if A **and** (B **or** C) **then** . . .

Bei gemischten Operatoren Klammern Pflicht

Kontrolldatenstrukturen

- **if then else**
- **case**
- **loop**
- **while loop**
- **for loop**

if then else

```
if Weather=Sunny then
   Happyness := 1000;
elseif Weather=Stormy
   Happyness := -1000;
else
   Happyness := 0;
end if;
```

if then else

```
if Weather=Sunny then
   Happyness := 1000;
elseif Weather=Stormy
   Happyness := -1000;
else
   Happyness := 0;
end if;
```

if then else

```
if Weather=Sunny then
   Happyness := 1000;
elseif Weather=Stormy
   Happyness := -1000;
else
   Happyness := 0;
end if;
```

case

```
case Range_in_Meter is
  when 1      => Jump;
  when 10..100 => Walk;
  when 101..10_000 => Take_Bike;
  when 7 | 77_777 | 1E9..Integer'Last => Beam;
  when others => Do_Nothing;
end case;
```

Auswahl muss erschöpfend und eindeutig sein

loop (1)

```
loop
 exit when Is_Empty(My_Stack);
 Cur_Item := Pop(My_Stack);
 Process_Item(Cur_Item);
end loop;
```

loop (2)

```
loop
 if Is_Empty(My_Stack) then exit; end if;
 Cur_Item := Pop(My_Stack);
 Process_Item(Cur_Item);
end loop;
```

while loop

```
while N < 10
loop
 Put(N);
 N := N + 1;
end loop;
```

for loop

```
Result:=1;
for I in 2 .. Faculty
 Result := Result*I;
end loop;
```

Laufvariable schreibgeschützt mit Sichtbereich Schleife

for loop

```
Result:=1;
for I in reverse 2 .. Faculty
 Result := Result*I;
end loop;
```

Laufvariable schreibgeschützt mit Sichtbereich Schleife

Arrays

- Arrayindex beliebig
- Arrayzugriffe werden normalerweise zur Laufzeit überprueft
- Arrays können per *pragma* gepackt werden
- Attribute: A'First A'Last A'Length
- Multidimensionale Arrays
- Zugriff: My_Field(X , Y)

Arrays

- Arrayindex beliebig
- Arrayzugriffe werden normalerweise zur Laufzeit überprueft
- Arrays können per *pragma* gepackt werden
- Attribute: A' First A'Last A'Length
- Multidimensionale Arrays
- Zugriff: My_Field(X , Y)

Beispiel

```
type Heap is array(1..100) of Float;

type Field is array(-100..100, -100..100) of
Integer;
```

Arrays

- Arrayindex beliebig
- Arrayzugriffe werden normalerweise zur Laufzeit überprueft
- Arrays können per *pragma* gepackt werden
- Attribute: A' First A'Last A'Length
- Multidimensionale Arrays
- Zugriff: My_Field(X , Y)

Beispiel

```
type Heap is array(1..100) of Float;

type Field is array(-100..100, -100..100) of
Integer;
```

Arrays Beispiel Bin Search

```
Left := A'First;
Right := A'Last;
while Left<Right
loop
 Middle := ( Left + Right ) / 2;
 exit when A(Middle) = Element;
 if (A(Middle) < Element ) then
    Left := Middle;
 else
    Right := Middle;
 end if;
end loop;
```

Arrays Beispiel Bin Search

```
Left := A'First;
Right := A'Last;
while Left<Right
loop
 Middle := ( Left + Right ) / 2;
 exit when A(Middle) = Element;
 if (A(Middle) < Element ) then
    Left := Middle;
 else
    Right := Middle;
 end if;
end loop;
```

Arrays Beispiel Bin Search

```
Left := A'First;
Right := A'Last;
while Left<Right
loop
 Middle := ( Left + Right ) / 2;
 exit when A(Middle) = Element;
 if (A(Middle) < Element ) then
    Left := Middle;
 else
    Right := Middle;
 end if;
end loop;
```

Programmeinheiten (Wiederholung)

Logische Einheiten

- subprogram
- package
- spezielle unit (taks unit, protected unit, generic unit)

Je Teilung in zwei Dateien

- Spezifikation (*.ads*)
- Implementation (*.adb*)

Programmeinheiten (Wiederholung)

Logische Einheiten

- subprogram
- package
- spezielle unit (taks unit, protected unit, generic unit)

Je Teilung in zwei Dateien

- Spezifikation (*.ads*)
- Implementation (*.adb*)

Subprogram Implementation

```
procedure
Average(One,Two : in Integer; Result: out Integer)

function
Average(One,Two : in Integer) return Integer
```

Subprogram Implementation

```
procedure
Average(One,Two : in Integer; Result: out Integer)

function
Average(One,Two : in Integer) return Integer
```

- `procedure` Methode ohne Rückgabewert
- `function` Methode mit einem Rückgabewert

Subprogram Implementation

```
procedure
Average(One,Two : in Integer; Result: out Integer)

function
Average(One,Two : in Integer) return Integer
```

- `procedure` Methode ohne Rückgabewert
- `function` Methode mit einem Rückgabewert
- `in` Parameter wird gelesen *call by value*
- `out` Parameter wird geschrieben *call by reference*
- `in out` Parameter wird gelesen und geschrieben *call by reference*

```
procedure Pascal is
   function Binomial(N:in Integer;K:in Integer) return Integer is
      type Pascal_Triangle is array(0..N,0..N) of Integer;
      A: Pascal_Triangle;
   begin
      for I in 0 .. N
      loop
         A(I,0):=1; A(I,I):=1;
         for J in 1..(I-1)
         loop
            A(I,J):=A(I-1,J-1)+A(I-1,J);
         end loop;
      end loop;
      return A(N,K);
   end Binomial;
   Input_N, Input_K : Integer;
begin
   Get(Input_N); Get(Input_K);
   Put(Binomial(Input_N,Input_K));
end Pascal;
```

OO Überblick

- Ada95: erste spezifizierte Sprache mit OO
- Ada2005: Modernisierung der OO Elemente
  ($\ominus$ Modular $\oplus$ Mehrfachvererbung (nur Interfaces))
- Eigenschaften:
  - Kapselung
  - dynamischer Polymorphismus
  - Vererbung (Signatur $+$ Implementation)

Objekt Definition

```
package Geometry is
  type Object is abstract tagged
   record
    X_Coord: Float;
    Y_Coord: Float;
   end record;

   function Area(O: Object) return Float
     is abstract;
end Geometry;
```

Objekt in Ada = `tagged type`

Vererbung

```
package Geometry.Circle is
   type Circle is new Object with
      record
         Radius: Float;
      end record;
   function Area(C: Circle) return Float;
end;
```

Vererbung

```
package Geometry.Circle is
   type Circle is new Object with
      record
         Radius: Float;
      end record;
   function Area(C: Circle) return Float;
end;
```

Ungewöhnlich: Das Objekt selbst muss mit übergeben werden (die Funktion *gehört nicht* dem Objekt)

Dispatching

- Ada95: `Geometrie.Circle.Area(My_Circle);`
- Ada2005: `My_Circle.Area;`

Ada2005: Erster Parameter kann weggelassen werden.

Dynamisches Dispatching (Ada2005):

```
type Pointer is access all Object'Class;
...
This_One: Pointer:=My_Circle'Access;
Put(This_One.Area);
```

`Object'Class` ist die Menge aller Object Klassen samt Unterklassen

Dispatching

- Ada95: `Geometrie.Circle.Area(My_Circle);`
- Ada2005: `My_Circle.Area;`

Ada2005: Erster Parameter kann weggelassen werden.

Dynamisches Dispatching (Ada95):

```
type Pointer is access all Object'Class;
...
This_One: Pointer:=My_Circle'Access;
Put(Geometrie.Circle.Area(This_One.all));
```

`Object'Class` ist die Menge aller Object Klassen samt Unterklassen

Interfaces (Ada2005)

- keine ”echte” Mehrfach-Vererbung (gefährlich)
- Interface Vererbung
- *siehe Java*

Interface Beispiel (1)

```
package P1 is
   type Computable is interface;
   function Value(X: Computable)
      return Float is abstract;
   procedure Simplify(X: Computable) is null;
end P1;
```

Interface Beispiel (1)

```
package P1 is
   type Computable is interface;
   function Value(X: Computable)
      return Float is abstract;
   procedure Simplify(X: Computable) is null;
end P1;
```

- Nur Schnittstellenvererbung

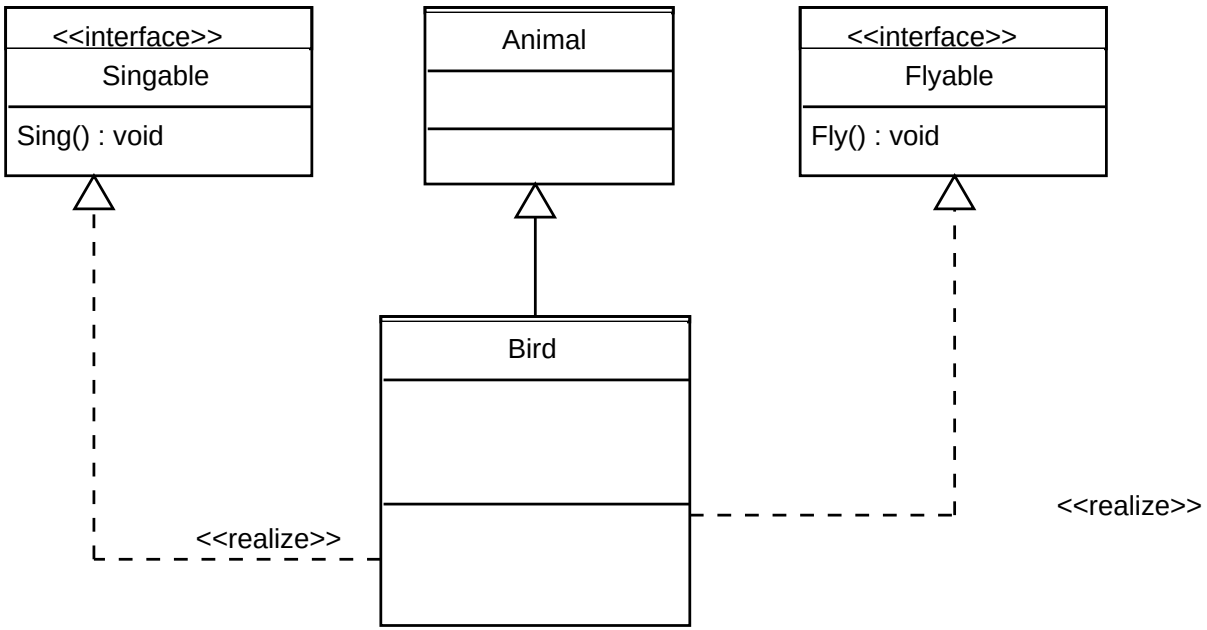
Interface Beispiel (1)

```
package P1 is
   type Computable is interface;
   function Value(X: Computable)
      return Float is abstract;
   procedure Simplify(X: Computable) is null;
end P1;
```

- Nur Schnittstellenvererbung
- Null Funktion

Interface Beispiel (2)

```
type Bird is new Animal and Flyable and Singable with
```



Initialize Adjust Finalize

Konstruktor und Destruktor (und Zuweiser ”:=”) überschreiben

```
type Controlled is abstract tagged private;
procedure Initialize(Object : in out Controlled);
procedure Adjust    (Object : in out Controlled);
procedure Finalize  (Object : in out Controlled);

type Limited_Controlled is abstract tagged limited private;
procedure Initialize(Object : in out Limited_Controlled);
procedure Finalize  (Object : in out Limited_Controlled);
```

(*Kein Grundobjekt wie in Java*)

Schutz overriding (Ada 2005) (1)

Beispiel

```
package Zoo is
   type Animal is new Limited_Controlled with...
   procedure Initialize(Obj: in out Animal);
   procedure Finialise(Obj: in out Animal);
end Zoo;
```

Finde den Fehler!

Schutz overriding (Ada 2005) (1)

Beispiel

```
package Zoo is
   type Animal is new Limited_Controlled with...
   procedure Initialize(Obj: in out Animal);
   procedure Finialise(Obj: in out Animal);
end Zoo;
```

Schutz overriding (Ada 2005) (1)

Beispiel

```
package Zoo is
   type Animal is new Limited_Controlled with...
   procedure Initialize(Obj: in out Animal);
   procedure Finialise(Obj: in out Animal);
end Zoo;
```

- Schlüsselwort **overriding** gibt an,dass eine Methode überschrieben wird
- Schlüsselwort **not overriding** gibt an, dass eine Methode *nicht* überschrieben wird

Schutz overriding (Ada2005) (2)

Beispiel mit overriding

```
package Zoo is
   type Animal is new Limited_Controlled with...
   overriding
   procedure Initialize(Obj: in out Animal);
   overriding
   procedure Finialise(Obj: in out Animal);
end Zoo;
```

$\Rightarrow$ Wirft Compilerfehler

(sonst schwierig zu entdecken)

Exceptions

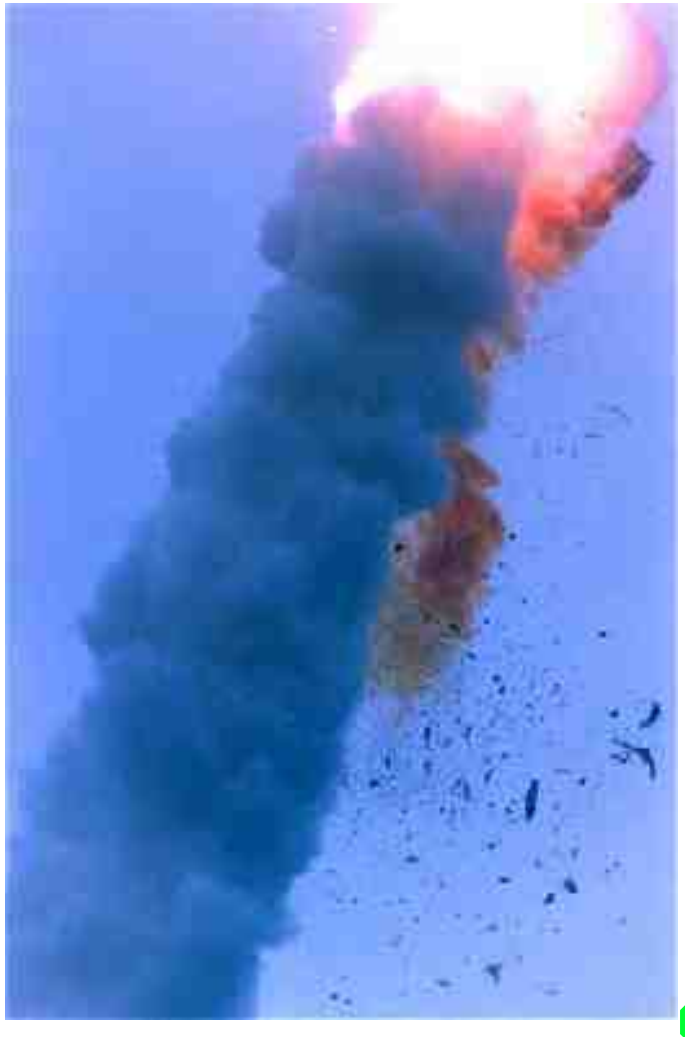
- Art der Fehlerbehandlung oft unklar (*Module*)
- Fehlerbehandlung durch Rückgabewerte kompliziert
- Ada bietet Exceptions
- Fehler $\rightarrow$ raise exception
- Standard: Programmabruch $+$ Fehlerausgabe
- Sonst: Fehler Abfangen und Behandeln

Exceptions: Schlechtes Beispiel (1)

- Absturz Ariane 5 durch eine Exception
- Übername Modul Ariane 4
- Überlauf: Cast 64 Bit Float $\rightarrow$ 16 Bit Integer

Exceptions: Schlechtes Beispiel (2)

- Nicht abgefangen (Ada Fehlerbehandlung wegen Performance abgeschaltet)
- Absturz Lenksystem
- Schieflage initiert Selbstzerstörung
- Schaden: 500 Mio $



Exceptions Definition

Definition

```
exception_declaration ::=
    defining_identifier_list : exception;
```

Beispiel

```
Overflow, Underflow:exception;
```

Ada Standard Exceptions:

- `Constraint_Error`
- `Program_Error`
- `Storage_Error`
- `Tasking_Error`

Exceptions Werfen (Ada2005)

Werfen

```
raise_statement ::= raise;
      | raise exception_name
      [with string_expression];
```

Beispiel

```
raise Underflow with
  "Error Pop_Element: Stack is empty";
```

Exceptions Behandeln

```
procedure Do_File_Operations() is
begin
  ...
exception
    when End_Of_File =>
        Close(Some_File);
    when File_Not_Found | File_Locked =>
        Text_IO.Put_Line("Could not open =(");
    when The_Error : others =>
        Put_Line("Unknown error:");
        Put_Line(Exception_Message(The_Error));
        raise;
end Do_File_Operations;
```

Generics

- *Vgl C++ Templates*
- Wichtige Hilfe um allgemeine Bibliotheken zu schreiben
- Typsicher
- Praktikable Fehlermeldungen zur Compilezeit
- Beispiel: Erstelle allgemeine Swap Funktion

Generics Beispiel: Swap (1)

Definition

```
procedure Swap(Left, Right : in out Integer);
```

Implementation

```
procedure Swap(Left, Right : in out Integer) is
   Temporary : Integer;
begin
   Temporary := Left;
   Left := Right;
   Right := Temporary;
end Swap;
```

Zu Speziell

Generics Beispiel: Swap (2)

Definition

```
generic
   type Element_Type is private;
procedure
Generic_Swap(Left, Right:in out Element_Type);
```

Implementation

```
procedure Generic_Swap
   (Left, Right : in out Element_Type) is
   Temporary : Element_Type;
begin
   Temporary := Left; Left := Right;
   Right := Temporary;
end Generic_Swap;
```

Generics Beispiel: Swap (3)

Instanziierung

```
procedure Swap_Test is
 procedure Swap is
   new Generic_Swap(Element_Type => Integer);
 A, B : Integer;
begin
 A := 5;
 B := 7;
 Swap(A, B);
 Put(A); -- Prints out 7
end Swap_Test;
```

Access Types

- Pointer sind gefährlich
- Pointer sind nützlich und effizient
- Pointer in Ada: Access Types
- wie C Pointer mit gewissen Sicherheitshilfen
- (eigentlich) keine Arithmetik

Access Types Definition (Objekt) (1)

Definition

```
access_type_definition:=
 "type" new_type_name
 "is" ["not" "null"] "access"
 ["constant"] ["all"] type_name ";"
```

Beispiel

```
type Access_Bird is access all Bird;
Favorite_Bird : Access_Bird;
...
begin
Favorite_Bird := Swan'Access;
...
Favorite_Bird := Stork'Access;
```

Access Types Definition (Objekt) (2)

Definition

```
access_type_definition:=
 "type" new_type_name
 "is" ["not" "null"] "access"
 ["constant"] ["all"] type_name ";"
```

- `all` Access Type kann auch nicht dynamisch erzeugte Objekte referenzieren
- `not null` Access Type darf nie Wert `null` annehmen
- `constant` referenzierter Wert darf nicht verändernt werden

Access Types - Vergleich C

Ada

```
type Node_Access access Node;
My_Node: Node;
Start,Current: Node_Access;
Current:=new Node;
Start:=My_Node'Access;
Current.all := Start.all;
Current.Data := 5;
```

Ansi C

```
typedef node *node_access;
node my_node;
node_access start,current = 0;
current = malloc(sizeof(node));
start = &my_node;
*current = *start;
current->data = 5;
```

- `.all` dereferenziert
- `'Access` gibt Referenz zurück
- `new` alloziiert Objekt

Alloziierung von Objekten (1)

- dynamische Erzeugung von Objekten
- `new Objekt_Name`
- notwendig: dynamische Vernichtung von Objekten
- automatisch (Garbage Collection)
  - $\oplus$ weniger Programmieraufwand
  - $\oplus$ weniger Programmierfehler
  - $\ominus$ signifikanter Performancenachteil
  - $\ominus$ Laufzeiteigenschaften unvorhersagbar
  - $\ominus$ erhöter Platzbedarf
- Compilerabhängig (Zb. J-Code Erzeugung (auch Applets))

Alloziierung von Objekten (2)

- manuelle Freigabe
  - ⊕ gute Peformance
  - ⊖ eröhter Programmieraufwand
  - ⊖ eröhte Fehlergefahr
- Ada Freigabe durch Generics Prozedur `Unchecked_Deallocation`

Beispiel Freigabe

```
type Tree_Access access Tree_Node;
Current : Tree_Access;
procedure Free is new
Unchecked_Deallocation(Tree_Node, Tree_Access);
...
Free(Current);
```

Freigabe trotz Gargabe Collection

Was passiert bei manueller Freigabe trotz Garbage Collection?

Freigabe trotz Gargabe Collection

Was passiert bei manueller Freigabe trotz Garbage Collection?

- Free Aufruf hat keine Auswirkungen (bzw wegoptimiert)
- allgemeinerer Stil: Mit Free Aufrufen

Gefährliche Automatische Freigabe

```
Current, Next : Access_Node;
procedure Free is new Unchecked_Deallocation(
   Tree_Node, Tree_Access);
Current := My_Node'Access;
Next := My_Node'Access;
...
Free(Current);
...
```

Gefährliche Automatische Freigabe

```
Current, Next : Access_Node;
procedure Free is new Unchecked_Deallocation(
   Tree_Node, Tree_Access);
Current := My_Node'Access;
Next := My_Node'Access;
...
Free(Current);
...
Next.Leaf;
```

Gefährliche Automatische Freigabe

```
Current, Next : Access_Node;
procedure Free is new Unchecked_Deallocation(
   Tree_Node, Tree_Access);
Current := My_Node'Access;
Next := My_Node'Access;
...
Free(Current);
...
Next.Leaf;
```

Resultierendes Verhalten undefiniert!

Lösung des Problems

Ada Design Problem: Sicher vs Effizient

Lösung:

- Vorsichtig mit Pointern umgehen
- Intelligente Pointer verwenden (langsamer)
- Garbage Collection (noch langsamer)

Tasks

- parallele Ausführung mehrerer Threads
- Wichtiger Bestandteil bei komplexen Softwareanwendungen
- Wichtiger Bestandteil in Echtzeitanwendungen
- GNAT unterstützt PULTS und KLT
- KLT's werden von nahezu allen bekannten Betreibssystemen unterstützt

Beispiel Task Spezifikation

```
task type Babbler is
   entry Start(Message : Unbounded_String;
               Count : Natural);
end Babbler;
```

Beispiel Task Body

```
task body Babbler is
  Babble : Unbounded_String;
  Maximum_Count : Natural;
begin
  accept Start(Message : Unbounded_String;
               Count : Natural) do
    Babble := Message;
    Maximum_Count := Count;
  end Start;
  for I in 1 .. Maximum_Count loop
    Put_Line(Babble);
    delay 1.0;
  end loop;
end Babbler;
```

Beispiel Task Body

```
task body Babbler is
   Babble : Unbounded_String;
   Maximum_Count : Natural;
begin
   accept Start(Message : Unbounded_String;
                Count : Natural) do
      Babble := Message;
      Maximum_Count := Count;
   end Start;
   for I in 1 .. Maximum_Count loop
      Put_Line(Babble);
      delay 1.0;
   end loop;
end Babbler;
```

Beispiel Task Body

```
task body Babbler is
   Babble : Unbounded_String;
   Maximum_Count : Natural;
begin
   accept Start(Message : Unbounded_String;
                Count : Natural) do
      Babble := Message;
      Maximum_Count := Count;
   end Start;
   for I in 1 .. Maximum_Count loop
     Put_Line(Babble);
     delay 1.0;
   end loop;
end Babbler;
```

Protected Types

Zwei Threads $\oplus$ gemeinsamen Daten $\Rightarrow$ Synchronisation
Ada bietet Protected Types zur Synchronisation:
Zugriff außschlieslich über:

- `protected function` (read only, simultanious)
- `protected procedure` (r/w, locked)
- `protected entry` (barrier)

Protected Entry Beispiel

```
protected type Resource is
  entry Seize;
  procedure Release;
private
  Busy : Boolean := False;
end Resource;
protected body Resource is
  entry Seize when not Busy is
  begin
    Busy := True;
  end Seize;
  procedure Release is
  begin
    Busy := False;
  end Release;
end Resource;
```

Protected Entry Beispiel

```
protected type Resource is
  entry Seize;
  procedure Release;
private
  Busy : Boolean := False;
end Resource;
protected body Resource is
  entry Seize when not Busy is
  begin
    Busy := True;
  end Seize;
  procedure Release is
  begin
    Busy := False;
  end Release;
end Resource;
```

Protected Entry Beispiel

```
protected type Resource is
  entry Seize;
  procedure Release;
private
  Busy : Boolean := False;
end Resource;
protected body Resource is
  entry Seize when not Busy is
  begin
    Busy := True;
  end Seize;
  procedure Release is
  begin
    Busy := False;
  end Release;
end Resource;
```

Interfacing to other languages

- größere Softwareprojekte nutzen oft mehrere Sprachen
- Verbindung zwischen den Sprachen notwendig
- Ada bietet hierfür folgende *pragma*s

```
pragma Import(C, getenv);
pragma Export(COBOL, Read_Sensor);
pragma Convention(Fortran, State_Vector);
```

Ada und Sicherheit

Auswahl einiger Sicherheitsfunktionen:

- 'Valid (Prüfe, ob eine Variable in seinem Wertebereich ist (Hardware Fehler, Kosmische Strahlung)
- pragma Normalize_Scalars (setze uninitialisierte Werte out-of-range)
- pragma restrictions (Untermenge von Ada, zb für SPARK)
- pragma assert(Boolean_Exp [,String_Exp]) (Laufzeitzusicherungen)
- pragma assertion_policy(Check | Ignor) (Laufzeitzusicherungspolitik)

Ada und Sicherheit

Auswahl einiger Sicherheitsfunktionen:

- 'Valid (Prüfe, ob eine Variable in seinem Wertebereich ist (Hardware Fehler, Kosmische Strahlung)
- pragma Normalize_Scalars (setze uninitialisierte Werte out-of-range)
- pragma restrictions (Untermenge von Ada, zb für SPARK)
- pragma assert(Boolean_Exp [,String_Exp]) (Laufzeitzusicherungen)
- pragma assertion_policy(Check | Ignor) (Laufzeitzusicherungspolitik)

weitere Sicherheitsfunktionen im Standard
optionale Implementierung im Compiler

Compileroptimierungen

Diverse Zusatzhilfen zur Codeoptimierung:

- pragma inline (inline kurze Funktionen)
- pragma optimize (auf Speed/Speicher)
- pragma suppress (keine Run-Time-Checks)
- pragma pack (Array Packen Bitweise zur Laufzeit)
- pragma Convention(Fortran, My_Array) (schneller Zugriff auf ersten arrayparameter)

Zusammenfassung

Programmiersprache Ada:

- Design auf Sicherheit
- Ada2005: modernes + robustes Konzept
- Kosten: gewisser erhöter Programmieraufwand

Weiterführende Literatur I


David A. Wheeler.
Ada 95: The Lovelace Tutorial - The Book
Springer-Verlag, 1997
http://www.adahome.com/Tutorials/Lovelace


John Barnes
Rational for Ada 2005
GNAT Pro Company, 2007
http://www.adacore.com/


ISO/IEC 8652:2007(E) Ed. 3
Ada Reference Manual *(Ada 2005)*
http://www.adaic.org/standards/05rm/html/RM-TTL.html