Introduction to Ada

Part 1: Basics of the Language

Lutz Berger

History of Ada

- Ada was developed under contract of the United Stated department of Defense (DoD) from 1977 to 1983
- Ada is influenced by Algol 68, Pascal and Modula-2
- Ada has influenced C++, Eiffel, Ruby, VHDL
- The name is derived from Ada Lovelace (1815–1852) who is called the first computer programmer
- The aim of the DoD was, to unify the languages used in the defense area, and it was mandatory to use it in this domain
- 1995 Ada 95 was released with object oriented (OO) support. Ada 95 was the first ISO-8652:1995 standard OO programming language

History of Ada

- 2005 came Ada 2005, with improvement of syntax for OO architecture, introducing containers similar to the STL library in C++, java like interfaces, EDF scheduling, timers and other features (ref. http://en.wikibooks.org/wiki/Ada_Programming/Ada_2005)
- With Ada 2012 aspects are introduced, which gives support to prove the correctness of implementation, support for multicore platforms (ref. http://en.wikibooks.org/wiki/Ada_Programming/Ada_2012)

Philosophy of Ada

- The philosophy of Ada is to provide most error detection at compile time and then detect what possible at runtime
- The concept of Ada’s syntax is to write out what is meant e. g “:=” for assignment and “=” for equal
- The language is designed for the real-time- , safety critical and parallel applications
- Ada is a platform independent language, which means the code once written can be compiled on many OS’s, specially RT systems like Vx-Works and Integrity.

Comparison to Main Stream

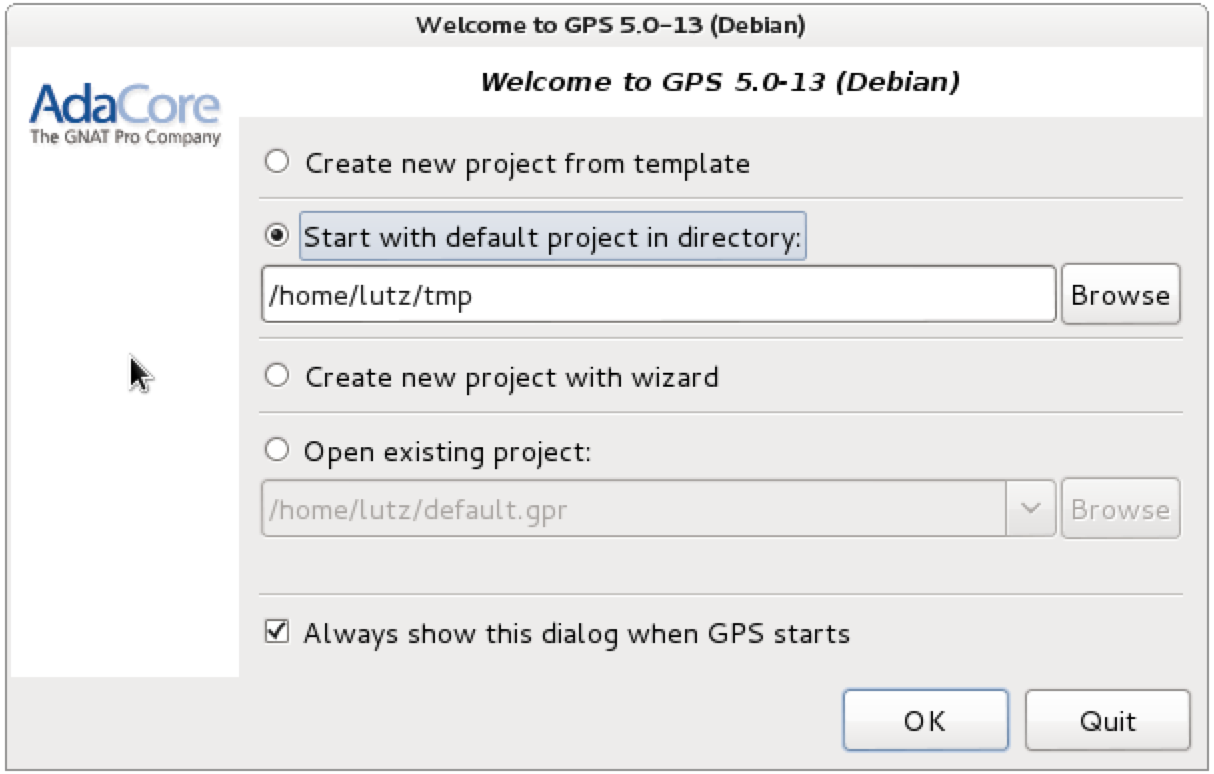
- Strong Type System
- Most possible errors are caught at compile time
- tasks are part of the language
- Real-time facilities are inbuilt
- Proof of correctness of implementation possible
- Easy restrict code for safety critical applications

Installing Ada on Debian

- For this course Debian is a good choice, since you can install the RT-Patch easily with the „apt-get install“ (as root) command for 64 Bit OS:
  - „apt-get install linux-image-3.2.0-4-rt-amd64“
  - „apt-get install linux-headers-3.2.0-4-rt-amd64“
- Realtime configuration for users: in /etc/security/limits.conf set rtprio
- Ada installation:
  - Download from http://libre.adacore.com/download/configurations
    - GNAT Ada GPL 2014
    - SPARK GPL 2014
    - Florist GPL 2014
  - Unpack the zip file.
  - In each component there is a tar.gz file, unpack them and
  - Run as root the ./doinstall script in each component. It will guide you through all the steps except in Florist GPL. Here use “./configure” and “make”, “make install” for installation

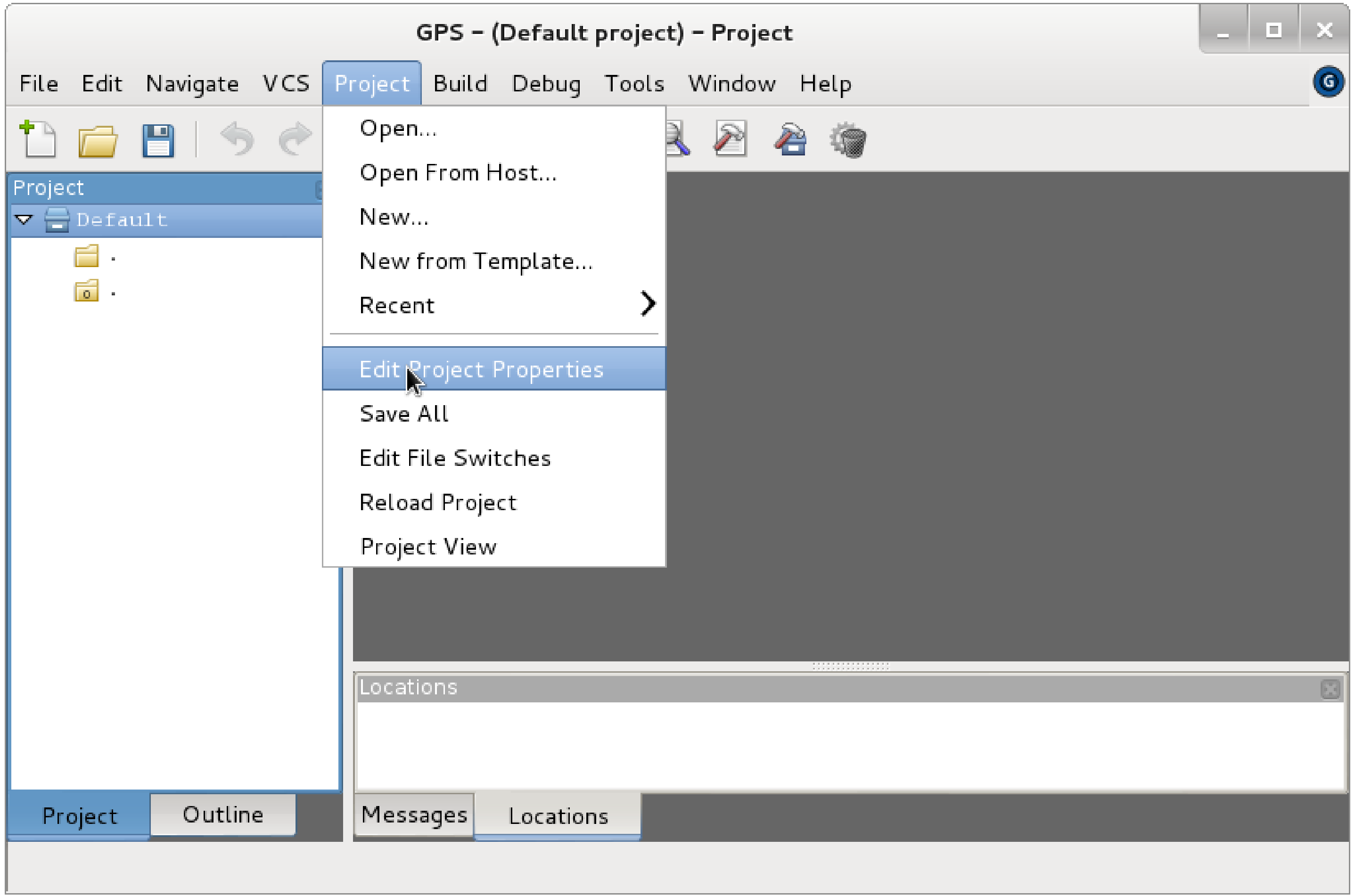
Hello World

- In a terminal go to a directory where you create your first Ada project and create a „src“ and „obj“ directory. Enter „gnat-gps&“



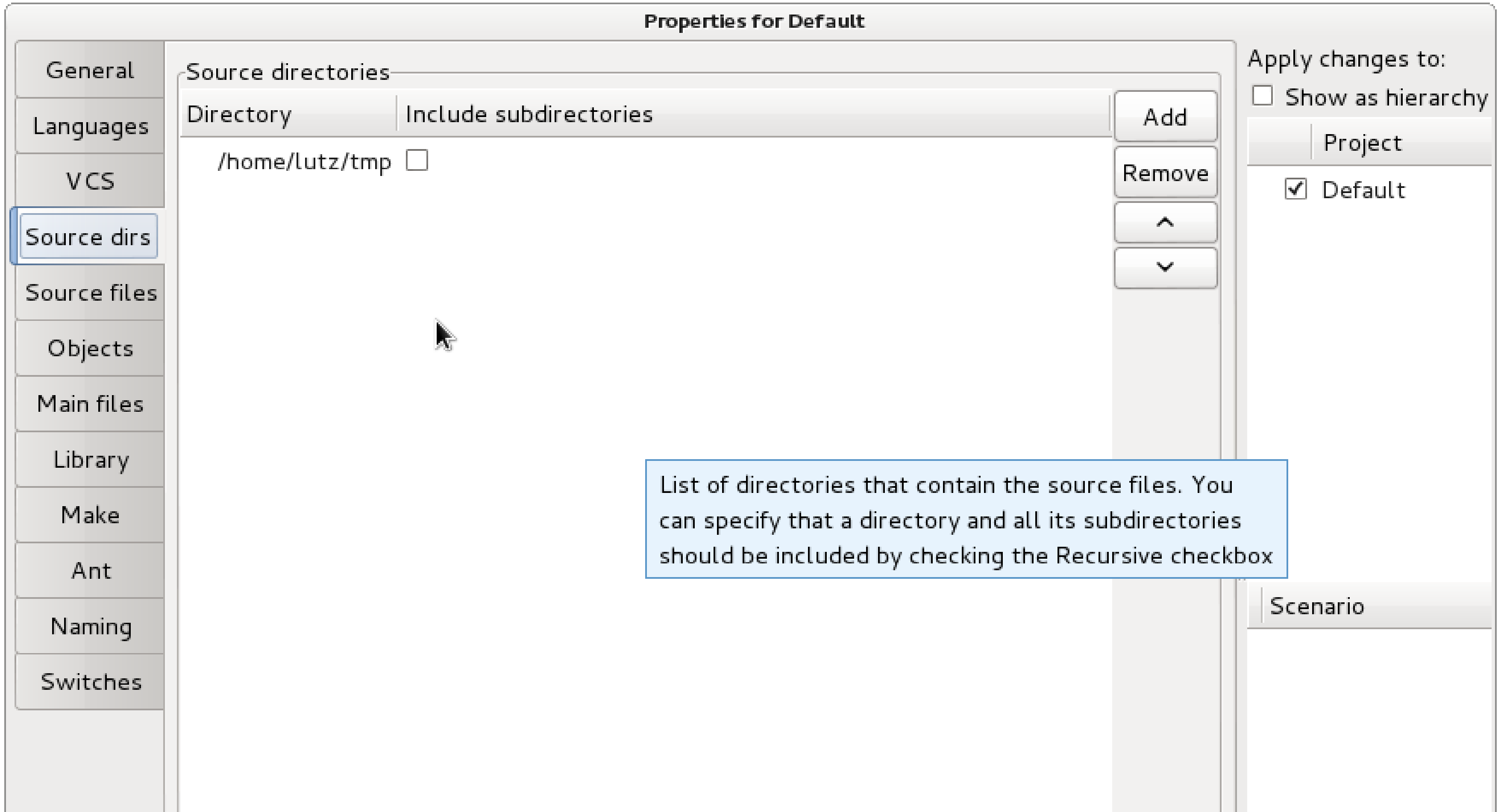
Hello World

- Select „Start with default project in directory“, enter „Ok“



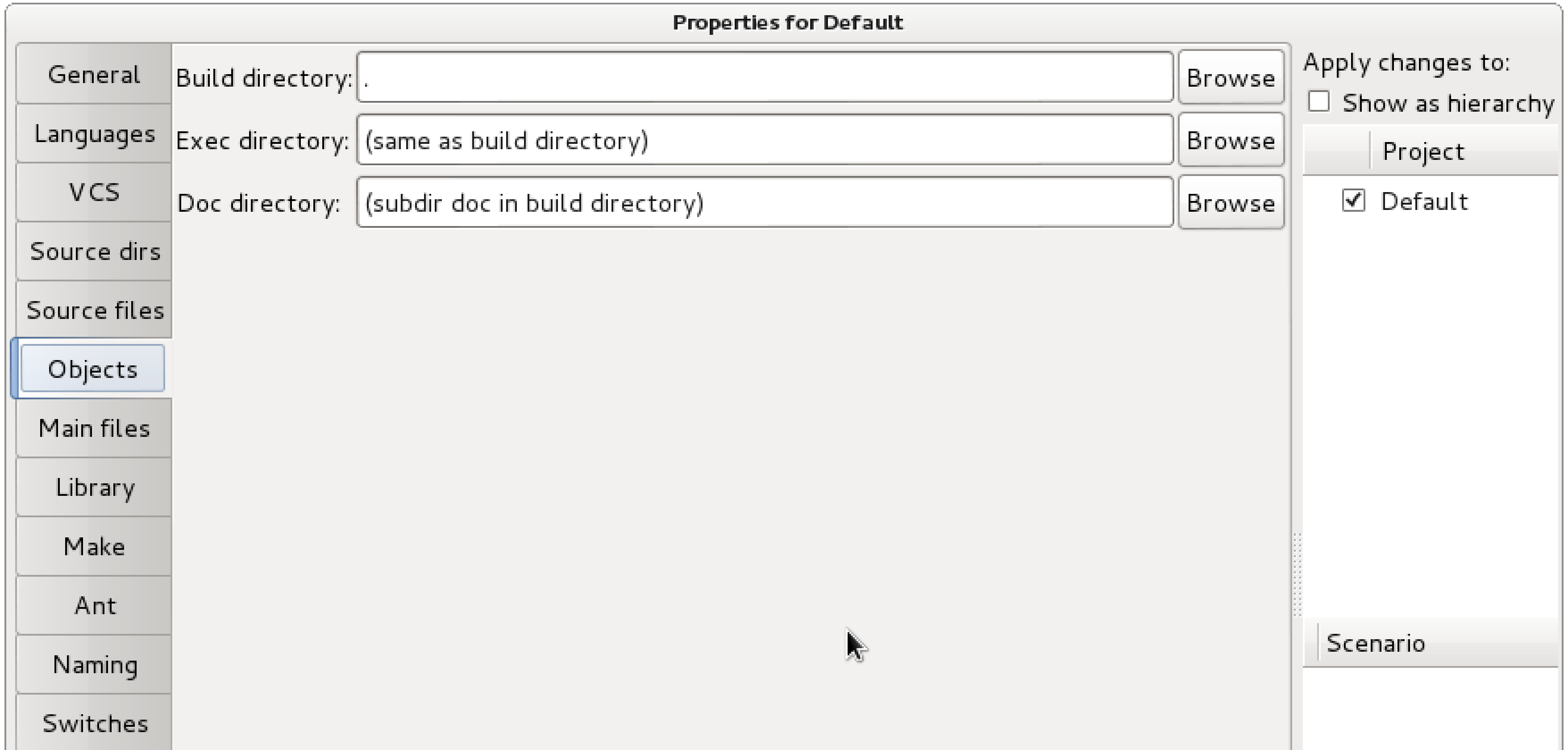
Hello World

- Select Project->Edit Project Properties



Hello World

- Remove the current source directory and add the „src“ dir, you have created before, select then the „Objects“ tab on the left



Hello World

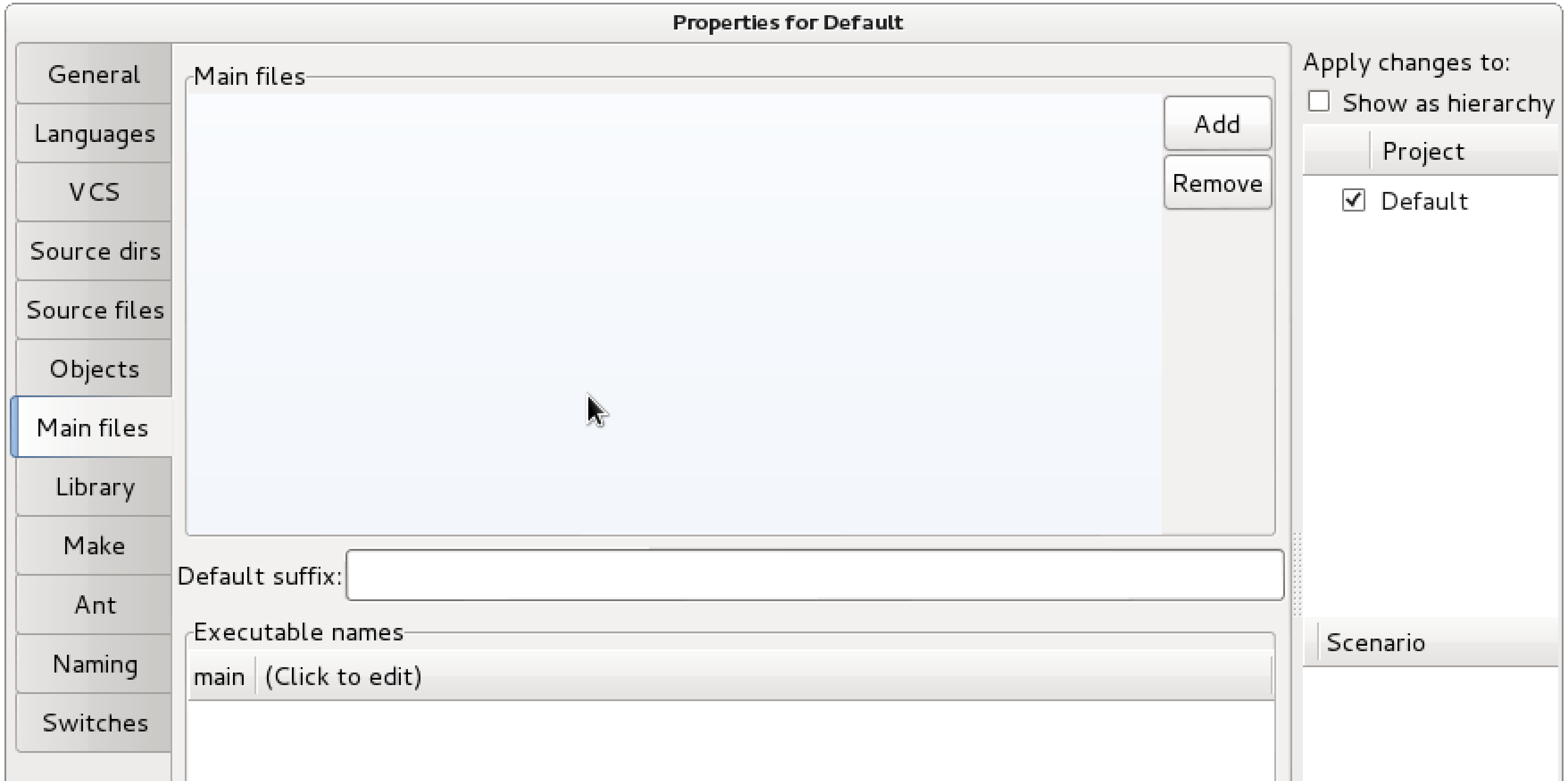
- On Build directory click on „Browse“ and select the „obj“ directory created before, then enter „Ok“ on the project property window
- Create a new file and save it as „hello.adb“ in the src directory
- Enter the following code:

```
with Text_IO;

procedure hello is
begin
   Text_Io.Put_Line("Hello world");
end;
```

Hello world

- Now we define the main entry file in the project in the project properties menu and add the just created „hello.adb“ file:



Hello World

- The „with“ statement includes code of „packages“ in the current compilation unit. Here the package from the Ada library „Text_IO“ is used to print the „Hello world“ string to the standard output
- Adding to the „with“ statement e.g. „use Text_IO;“ would make the call to the output procedure „Put(„Hello world“);“ sufficient.
- With the project Properties we have already separated the source and build environment. If you save the project in the „Project“ Menu with „save all“, the gnat project file will be updated. In our example it is „default.gpr“

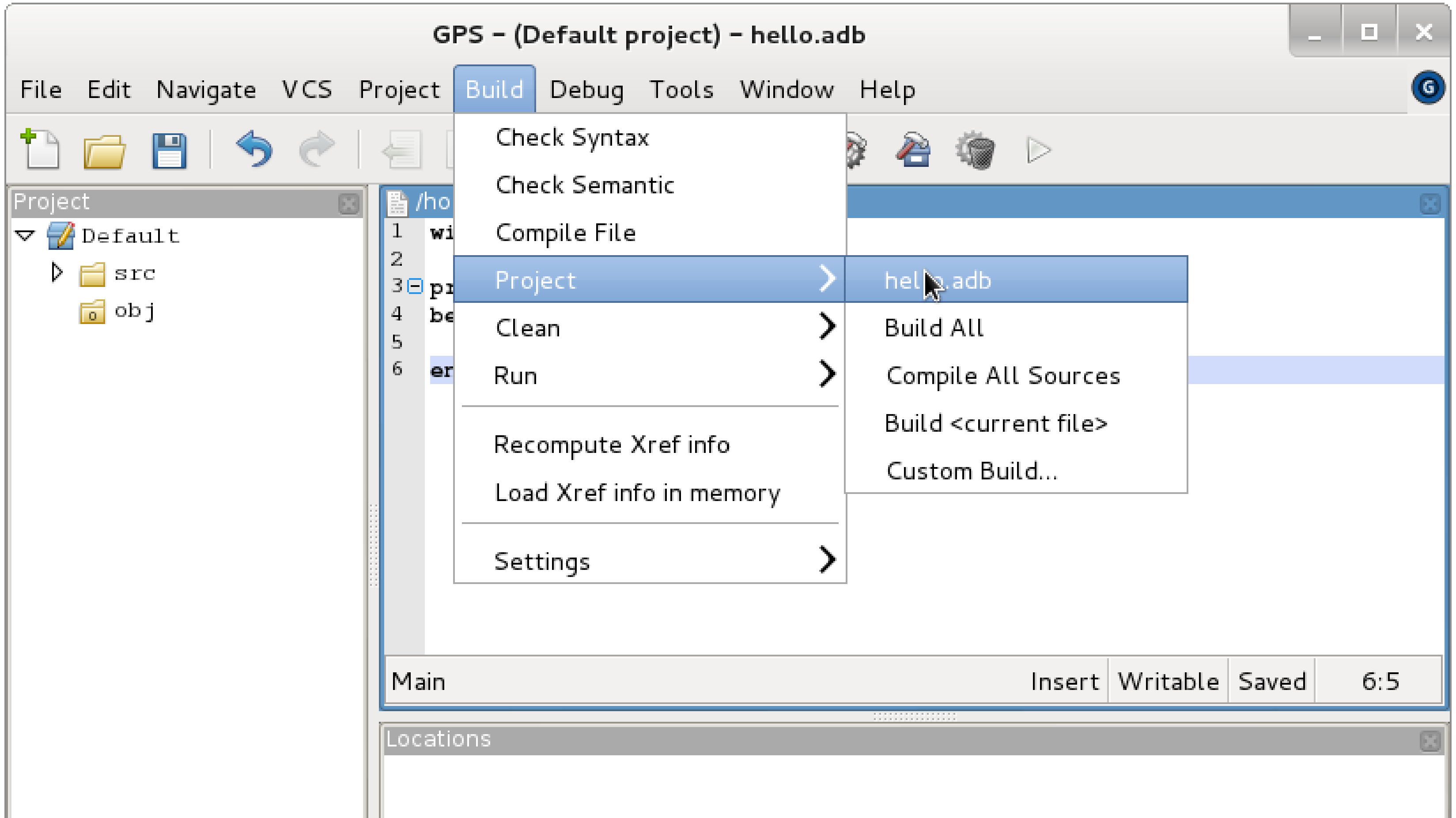
Hello World

- Here is the created default.gpr file:

```
project Default is
   for Source_Dirs use ("src");
   for Object_Dir use "obj/";
   for Main use ("hello.adb");
end Default;
```

Building Hello world

- Buiding the project you go to Build->Project->hello.adb



Building and Running Hello World

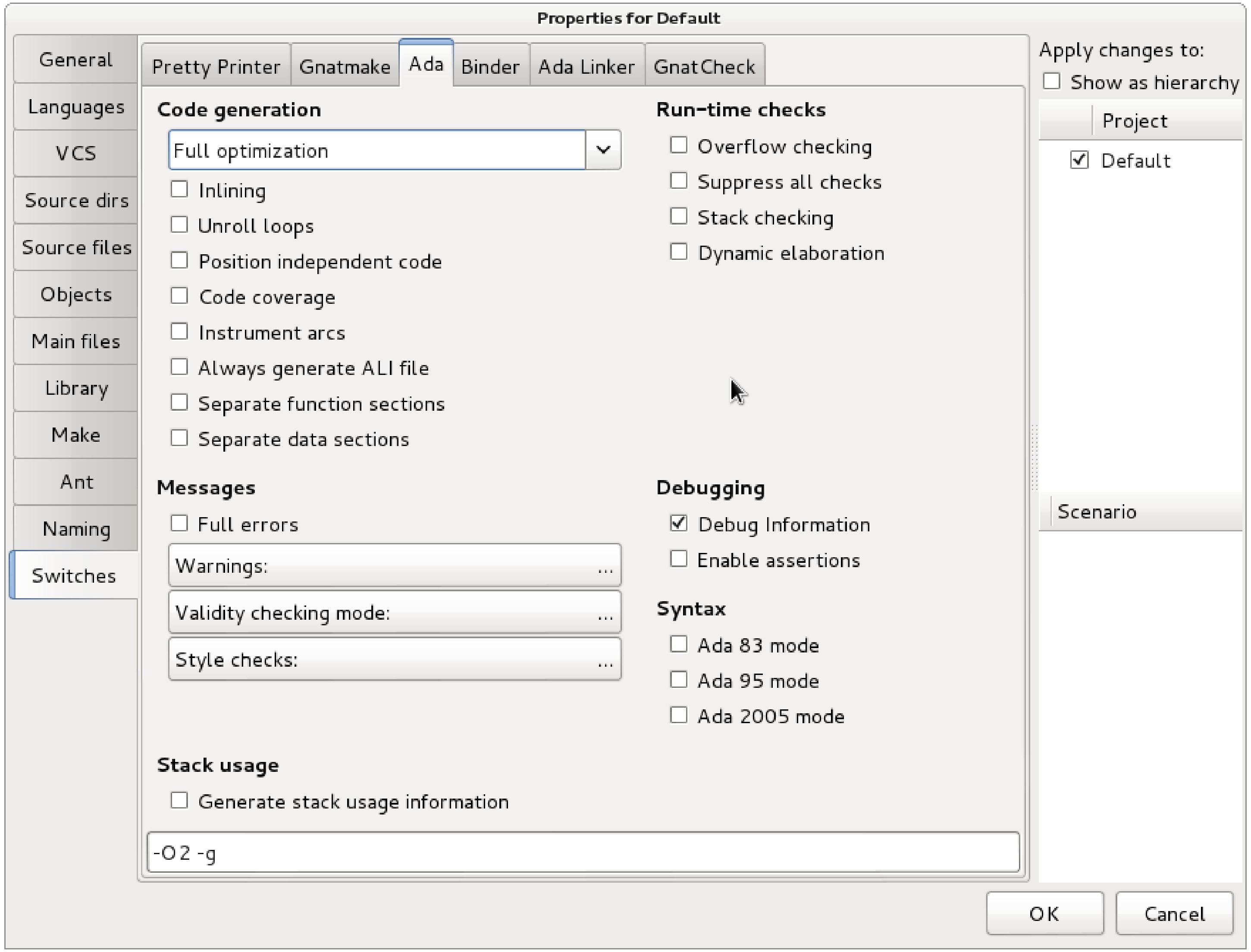
- Just click „Execute“ on the dialog popping up, the project will be build
- Down in the Message Window you can see the built result, if warnings and errors occurs, you can easily navigate to the respective code location
- Repeat the same with „Build->Run, in the Message window you will see the output „Hello world“ as the application is programmed.

Gnat Project File

- As we have seen, the „.gpr“ file is the gnat project file, where information about all the software project is configured. In Project Properties you can also set many other properties. We show now an example of compile switches and linker options.
- Open Project->Edit Properties again
- Click the „Switch“ tag on the left
- Now on the Ada tab select „full optimization“ and check „debug information“ like shown in the following screen shots

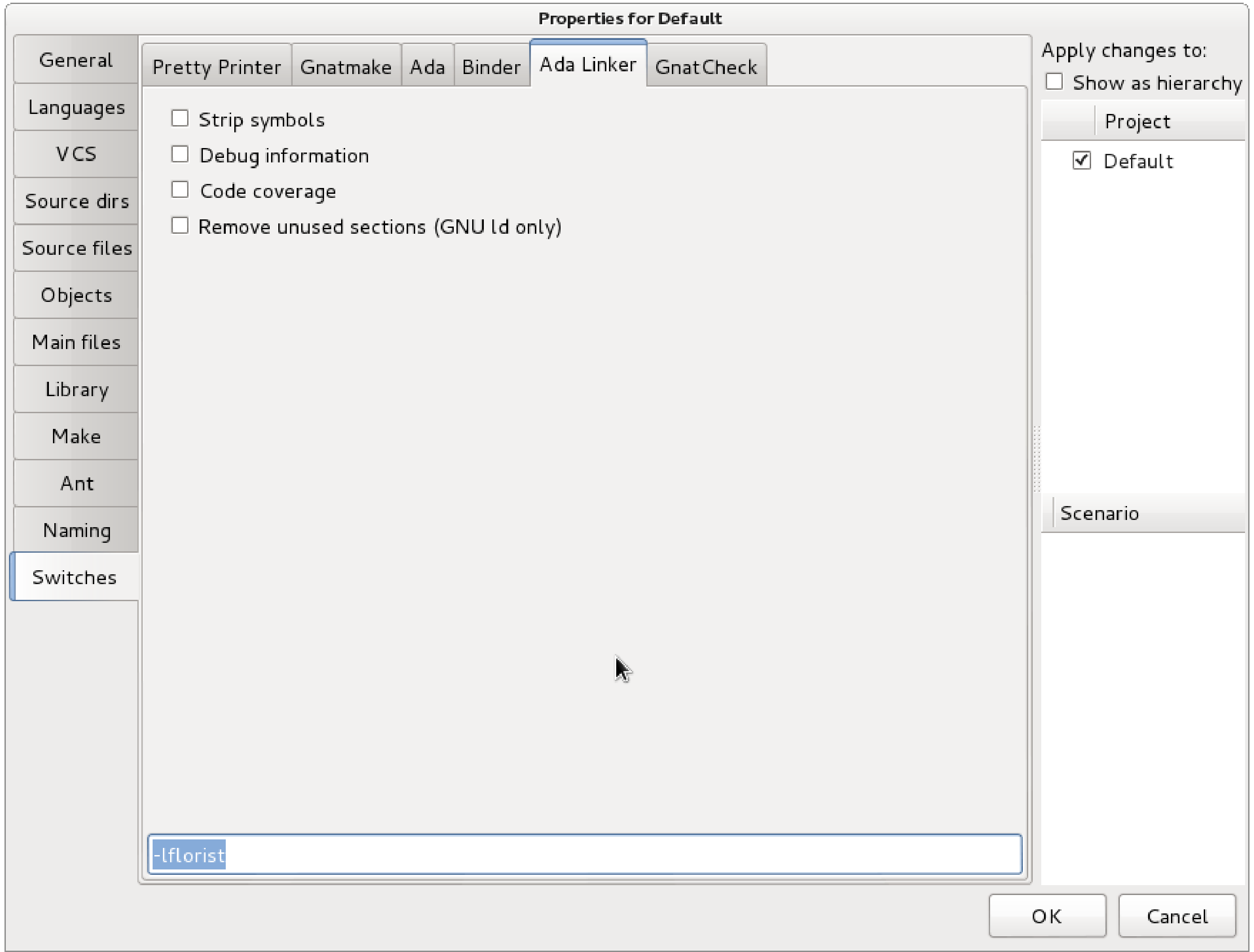
Gnat Project File

- Ada Switches



Gnat Project File

- Ada Linker Switches



Gnat Project File

- Click „Ok“ and save the project with the Menu „Project->Save All“
- The „default.gpr“ is generated with two additional entries:
  - Package Compiler and package Linker

```
package Compiler is
     for Default_Switches ("ada") use ("-O2", "-g");
  end Compiler;

  package Linker is
     for Default_Switches ("ada") use ("-lflorist");
  end Linker;
```

Command-Line Build

- If you want to embed the build process into an automated build environment, e. g. continuous integration, it is useful to trigger the build from command line. Here the command „gnatmake“ is used
  - Without any arguments, gnatmake gives a brief info about its arguments
  - With „gnatmake –Pdefault.gpr“ the project is build with the defined project properties
  - If the project file has no main function defined, „gnatmake <Main> -P<Project file> build the executable <Main>
  - The command „gnatclean“ is used to clean up the project

Typsystem in Ada

- The basic types: „Integer, Float, Duration, Character, String, Boolean“
- Further see http://en.wikibooks.org/wiki/Ada_Programming/Type_System
- For platform independence, the package „Interfaces“ is very useful, since here are Types with fixed lengths declared, e. g. „Interfaces.IEEE_Float_64“
- Example of enum type:

```
type On_Off_Type is (On, Off);
On_Off: On_Off_Type := On;
```

User Defined Types

- You can have any user defined types, examples

```
type Binary is range 0 .. 1;
type Hex is range 0 .. 15;
type Byte is mod 2**8;
```

  - The mod keyword implements a wrap around arithmetic
- You can not mix different types:

```
h: Hex := 1;b: Binary := h;
```

Will cause a compilation error of type mismatch

Access Types

- Pointer types in Ada are „Access types“ there are 3 different types of pointers
  - General „read write“ Access
    - Syntax: `type <ACCESS_ITEM_TYPE> is access all <ITEM_TYPE>`
      - This pointer can be assigned to an instances of the <item_type>‘s Access Attribute (Attributes see below)
      - It can‘t be assigned to a pointer of a constant
  - „read write“ Access
      - Syntax: `type <ACCESS_ITEM_TYPE> is access <ITEM_TYPE>`
      - Same as above, assignment to Access attribute not allowed
  - constant “read only“ Access
    - Syntax: `type <ACCESS_ITEM_TYPE> is access constant <ITEM_TYPE>`
      - Same as General Access, assignment to constant pointers allowed, their content can’t be changed

Access Types

- Here an example

```
   type Hex is range 0 .. 15;
   type Hex_Read_Access is access Hex;
   type Hex_ReadWrite_Access is access all Hex;
   type Hex_Constant_access is access constant Hex;

   package Hex_Text_Io is new Ada.Text_IO.Integer_IO (Hex);
   h: aliased Hex := 1;
   Null_Hex: aliased constant Hex := 0;
   hPtr_Read_Only : Hex_Read_Access := new Hex;
   hPtr_ReadWrite : Hex_ReadWrite_Access := h'Access;
   hPtr_Constant : Hex_Constant_Access := Null_Hex'Access;
   i: Integer := 234;

begin
   h := 10;
   hPtr_Read_Only.all := h;
   Put("the value at read only hex pointer is: "); --use Text_IO
   Hex_Text_IO.Put(hPtr_Read_Only.all);
   New_Line; --use Text_IO
   h := 5;
   Put("the value at read write hex pointer is: "); --use Text_IO
   Hex_Text_IO.Put(hPtr_ReadWrite.all);
   New_Line; --use Text_IO
   Put("Integer is: :"); --use Text_IO
   Put(i); --use Ada.Integer_Text_IO
end;
```

Access Type Example

- Explanation of code:
  - Line 8 – 10: Pointer types declarations accessing Hex Type
  - Line 12: We want to print out values of type Hex, thus we derive a custom package from the build in generic ada package „Ada.Text_IO.Integer_IO“
    - Short excursion to generics:

```
generic
   Type Item_Type is private
procedure/function (Item: Item_Type); -- one or      more declarations
```

    - In the code line the Item_Type is assigned to our Hex type with the new statement
  - Line 13: Any Type instances are created with the „Access“ attribute if declared with the „aliased“ statement
  - Line 15: With the „new“ statement here, new Memory is allocated for the Hex Type and the pointer hPtr_Read_Only points to it

Access Type Example

- Line 16-17: Here the pointers are assigned to the pointers of „h“ and „Null_Hex“. To be able to read the pointer value of this variables, the „aliased“ keyword was used in their declaration
- Line 22: „.all“ is dereferencing the pointer
- Line 30-31: Since the use statement is used, the full package has not to be specified. Ada automatically dispatches the correct package from the procedure/function signature

- The output of the Program:

```
the value at read only hex pointer is:  10
the value at read write hex pointer is:   5
Integer is: :        234
```

Attributes

- With Attributes you can get or set properties of language entities
- Attributes are retrieved with an apostrophe <ENTITY>‘<ATTRIBUTE>
  - Example „Integer‘First“ give the first value of the Type Integer, „Integer‘Last“ the last value
- To set an attribute the „for“ clause is used:
  - for <ENTITY>‘<ATTRIBUTE> use <VALUE>
  - Example:

```
type byte is range -128 .. 127;
for byte‘size use 8; -- the compiler must use 8 bits
```

- Further reading: http://en.wikibooks.org/wiki/Ada_Programming/Attributes

Packages

- As the software code is growing, we don‘t want to have all code in one file, but spread it over different and organize it. In Ada we have already used in-build packages e. g. Text_IO for text output/input.
- Packages are spread over specification- and body files ending with „ads“ and „adb.
  - package spec:

    ```
    package <Package Name> is
    -- further declarations
    end package <Package Name>
    ```
  - package body:

    ```
    package body <Package Name> is
    -- further implementations
    end package <Package Name>
    ```

Packages

- Declarations in the spec part are visible in other packages
- Declarations and definitions in the body part are only visible within the body
- Example:

```ada
package Array_Calculations is

    -- we use subtype here, because we want to be able to use the index in the
    -- assingment to the array elements.
    subtype Arr_Index is Integer range 1 .. 10;

    -- array type decleration
    -- it is save to use the Arr_Index with Range attribute everywhere where we
    -- loop through the array.
    type Int_Arr_Type is array (Arr_Index'Range) of Integer;

    function Sum( Arr: in Int_Arr_Type) return Integer;
end Array_Calculations;
```

Packages

```
package body Array_Calculations is

   function Sum( Arr: in Int_Arr_Type) return Integer is
      s: Integer := 0;
   begin
      for i in Arr_Index'Range loop
         s := Arr(i) + s;
      end loop;
      return s;
   end Sum;
end Array_Calculations;
```

Packages

```
with Ada.Integer_Text_IO;
with Text_IO;
with Array_Calculations;
use Array_Calculations;
procedure Package_Example is
   Int_Arr : Int_Arr_Type := (others => 0); -- initialize all elements to 0
begin
   for i in Arr_Index'Range loop
      Int_Arr(i) := i;
   end loop;

   -- with the declare statement you can declare any new variables and types
   -- at any non declerative part of code. They are only visible between the
   -- declare ... end block after the declare statement
   declare
      s: Integer := 0;
   begin
      s := Sum(Int_Arr);
      Ada.Integer_Text_Io.Put(s);
      Text_IO.New_Line;
   end;
end Package_Example;
```

The Ada Real-time Package

- The most important function here is „Clock“ retrieving the current time with the resolution of the processors cycle.
- „Clock“ returns a „Time“ type, derived from the type „Duration“
- „Duration“ is an ordinary fixed point type with a resolution of 1 ns
  - e. g. could be implemented as:

```
type Duration is delta 10.0**(-9)
        range -9223372036.854775808 .. 9223372036.854775807;
for Duration'Small use 10.0**(-9);
```

  - further reading about ordirnary fixed point see http://en.wikibooks.org/wiki/Ada_Programming/Types/delta

Ada Realtime Package

- We have conversion functions from type „Time_Span“ to „Duration“ and vice versa
- The „Split“ procedure spits the time into count seconds and time precision within one second, „Time_Of“ does the reverse
  - procedure Split (T : Time; SC : out Seconds_Count; TS : out Time_Span);
  - function Time_Of (SC : Seconds_Count; TS : Time_Span) return Time;
- The Minutes and Seconds(Ada 2005), Milliseconds, Microseconds and Nanoseconds are functions returning Time_Spans with the defined time interval
- We have overloaded +, -, *, / and comparison operation which makes time calculations possible.

The Delay Statement

- 2 versions of the delay statement:
  - delay <Duration>
    - delays the current thread for <Duration> seconds. <Duration> must have a decimal point even if it is a round second number
      - delay 1   -- not allowed
      - delay 1.0 -- correct
  - delay until <Time>
    - delays the current thread until the absolute time <Time>, if <Time> is in the past, „delay“ returns immediately

Real-time Clock example

```
with Ada.Real_Time;
use Ada.Real_Time;

with Text_IO;
procedure Main is
   curr_time : Time := Clock;
   s: Seconds_Count;
   d: Duration;
   Hundred_Milliseconds : constant Time_Span := Milliseconds(100);

   procedure Print_Time(m: String; t: Time) is
      ts: Time_Span;
   begin
      Split(t, s, ts);
      d := To_Duration(ts);
      Text_IO.Put_Line(m & s'Img & "s and" & d'img & "s" );
   end;

begin
   Print_Time("Time_First is: ", Time_First);
   Print_Time("Now is: ", curr_time);

   for i in 1 .. 10 loop
      delay To_Duration(Hundred_Milliseconds);
      curr_time := Clock;
      Print_Time("Now is with delay: ", curr_time);
      delay 0.01; -- simulate some intensive calculations
   end loop;
```

Real-time Clock example

```
declare
   next_time: Time := Clock + Hundred_Milliseconds;
begin
   next_time := Clock + Hundred_Milliseconds;
   for i in 1 .. 10 loop
      delay until next_time;
      next_time := next_time + Hundred_Milliseconds;
      Print_Time("Now is with delay until is: ", Clock);
      delay 0.01; -- simulate some intensive calculations
   end loop;
end;
end;
```

- In this example we use the local procedure “Print_Time” to print the time. We use the “delay” and “delay until” statement and simulate some intensive calculations.

Real-time Clock example

- output of the Example:

```
Time_First is: -9223372037s and 0.145224192s
Now is:  1408096514s and 0.501042000s
Now is with delay:  1408096514s and 0.601397000s
Now is with delay:  1408096514s and 0.711847000s
Now is with delay:  1408096514s and 0.822339000s
Now is with delay until is:  1408096514s and 0.933806000s
Now is with delay until is:  1408096515s and 0.033789000s
Now is with delay until is:  1408096515s and 0.133831000s
```

- Here the timing after the “delay until” statement is accurate with a precision of about 1 ms, whereas with the “delay” case you would have to correct the delay time with the execution time in the loop

Task Type

- Creating task in Ada is very easy
  - Declare a task type and a according body

```
task type <Name of Task Type>;
task body <Name of Task Type> is
begin
      -- code of task
end <Name of Task Type>
```

  - Define the task

```
<Name of Task>: <Name of Task Type>
```

  - If in the declaration the keyword “type” is omitted, it is also defined a an anonymous task

Execution of Static Tasks

- After the declarative part of the task is elaborated, the task is created.
- If the task is defined on library level, which means not having a parent task, it is activated after its definition
- If the task is having a parent, it is activated after the “begin” statement of the parent

Simple Task Example

```
with Text_IO;

procedure Main is
   task type simple_task_type;
   task body simple_task_type is
   begin
      Text_Io.Put_Line("Hello from your first task");
   end;

   task anonymous_task;
   task body anonymous_task is
   begin
      Text_Io.Put_Line("Hello from anonymous");
   end;

   simple_task: simple_task_type;
begin -- tasks start diract after begin statement
   null;
end;
```

Simple Task Example

- In the above example, the main routine is doing nothing than activating its tasks after the begin statement.
- The “null” statement is only executed after all tasks have finished their activation
- At the end Statement the main routine waits until all the child tasks have finished their execution

Dynamic Tasks

- Dynamic tasks are created with the “new” keyword
  - Declare a task type and a according body

```
task type <Name of Task Type>;
task body <Name of Task Type> is
begin
      -- code of task
end <Name of Task Type>
```

  - Define the pointer type to the task type

```
type <Name of Task Pointer> is access <Name of Task Type>
```

  - create and activate the task dynamically:

```
<Task Pointer> : <Name of Task Pointer> := new <Name of Task
Type>
```

Discriminants in Tasks

- With discriminants you can pass parameters to the task declaration, even pointers as in the example line 8
  - In line 6 an array type is declared with undefined range “<>”. In line 10 an instance is created with the “new” keyword and defined range.

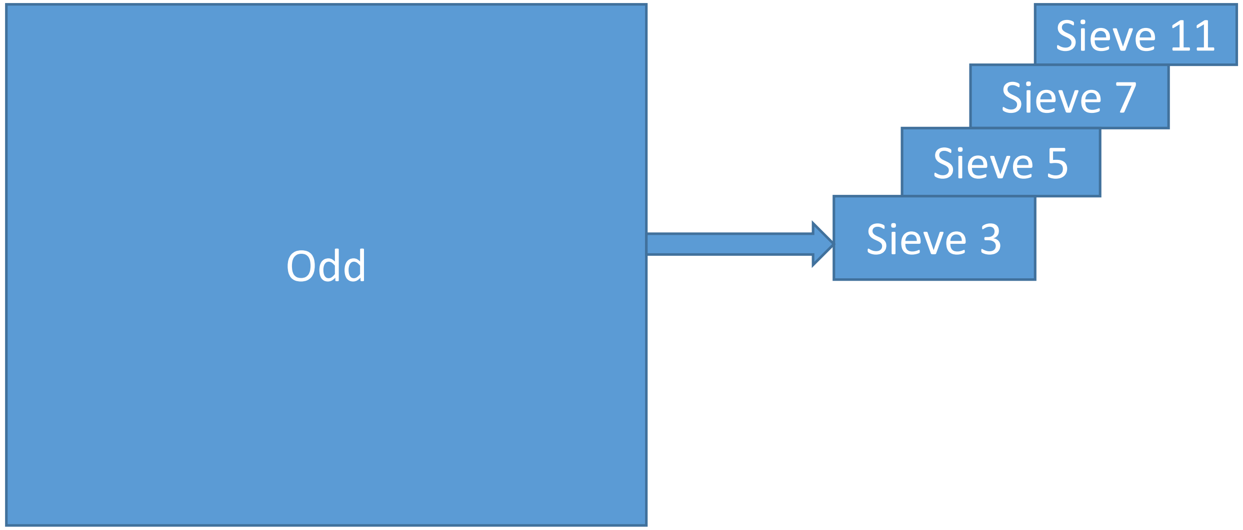
```
1  with Ada.Numerics.Discrete_Random;
2  with Text_IO;
3  with Ada.Integer_Text_IO;
4  procedure main is
5
6     type AI is array (Positive range <> ) of Integer;
7     type PAI is access all AI;
8     task type Exchange_Sorter(A : access AI);
9     subtype One_Hundred is Integer range 1 .. 100;
10    X : PAI := new AI (One_Hundred'Range); -- Initialised elsewhere
11    type Exchange_Sorter_Ptr_Type is access Exchange_Sorter;
12    Sorter_Ptr : Exchange_Sorter_Ptr_Type;
13    package Random is new Ada.Numerics.Discrete_Random(Positive);
14    RNG: Random.Generator;
```

Discriminants in Tasks

```
task body Exchange_Sorter is
   Sorted : Boolean := False;
   Tmp : Integer;
begin
   Text_Io.Put_Line("Exchange Sorter task running");
   while not Sorted loop
      Sorted := True;
      for I in A'First .. A'Last - 1 loop
         if A(I) < A(I + 1) then
            Tmp := A(I);
            A(I) := A(I + 1);
            A(I + 1) := Tmp;
            Sorted := False;
         end if;
      end loop;
   end loop;
   Text_Io.Put_Line("Exchange_Sorter finished");
end Exchange_Sorter;
```

Sieve Example

- As a last example for this lecture an example for evaluating prime numbers is given the so called sieve Example



Sieve Example

- The algorithm creates an "Odd" task, which delivers a series of odd numbers, creating an initial "Sieve task"
- Each Sieve task saves its "Prime" number, retrieves further numbers from the "Buffer" (the Buffers "Get" waits for the "Put"), throws them away if dividable by its prime number and creates a new sieve task, by this way a chain of sieve tasks are created.
- The application doesn't compile yet, since the Buffer is not yet defined. It will also not terminate properly. Later we will come back the complete example
- In the next lecture we will learn more about synchronization of task

Sieve Example

```
procedure Main is
   task type Sieve(B: access Buffer);
   type Sieve_Ptr is access Sieve;

   function Get_New_Sieve(B: access Buffer)
                           return Sieve_Ptr is
   begin
      return new Sieve(B);
   end Get_New_Sieve;

   task Odd;
   task body Odd is
      Limit : constant Positive := 1000;
      Num: Positive;
      Buf: aliased Buffer;
      S: Sieve_Ptr := Get_New_Sieve(Buf'Access);
   begin
      Num := 3;
      while Num < Limit loop
         Buf.Put(Num);
         Num := Num + 2;
      end loop;
   end Odd;
```

Sieve Example

```
task body Sieve is
   New_Buf : aliased Buffer;
   Next_Sieve : Sieve_Ptr;
   Prime, Num: Natural;
begin
   B.Get(Prime);
   -- Prime is a prime number, which coud be output
   loop
      B.Get(Num);
      exit when Num rem Prime /= 0;
   end loop;

   Next_Sieve := Get_New_Sieve(New_Buf'Access);
   New_Buf.Put(Num);

   loop
      B.Get(Num);
      if Num rem Prime /= 0 then
         New_Buf.Put(Num);
      end if;
   end loop;
end Sieve;

begin
   null;
end;
```

Summery

- This lecture we have learned:
  - how to install a Linux real-time OP, gnat compiler
  - a first “hello world” application and explored here some configurations of the “Gnat Project”, how to organize the source and build environment, set compiler and linker options
  - the basics of the type system of Ada specially user define types and access types
  - the concept of attributes
  - organizing the code in packages
  - Real-time abilities
    - the Ada.Real_Time package
    - “delay statement

Summary

- Tasks
  - static tasks
  - dynamic tasks
  - task discriminants
  - finally the “sieve” example for prime number calculation
- Homework
  - http://en.wikibooks.org/wiki/Ada_Programming/Control
  - http://en.wikibooks.org/wiki/Ada_Programming/Type_System
  - http://en.wikibooks.org/wiki/Ada_Programming/Types/record
  - http://en.wikibooks.org/wiki/Ada_Programming/Attributes
  - http://en.wikibooks.org/wiki/Ada_Programming/Subprograms
  - http://en.wikibooks.org/wiki/Ada_Programming/Packages
  - finish the online course http://university.adacore.com/courses/programming-in-the-large1/
- Literature: **Concurrent and Real-Time Programming in Ada,** ISBN: 9780521866972