SubTypes and Derived Types
Arrays, Records

Strongly typed

- Ada is strongly typed - different types of the same family are incompatible with each other;
- a value of one type cannot be assigned to a variable from the other type
- "mixed mode" expression trigger a compilation error
- type conversions must be made explicit

```
type Meters is new Float;
type Miles is new Float;
Dist_Imperial : Miles;
Dist_Metric : constant Meters := 100.0;
 Dist_Imperial := (Miles (Dist_Metric) * 1609.0) / 1000.0; --
Type conversion, from Meters to Miles
```

Type conversion

```
 function To_Miles (M : Meters) return Miles is  begin
return (Miles (M) * 1609.0) / 1000.0;   end To_Miles;

Dist_Imperial : Miles;

Dist_Metric   : constant Meters := 100.0;begin
Dist_Imperial := To_Miles (Dist_Metric);

Put_Line (Miles'Image (Dist_Imperial));
```

Derived types

- you can create new types based on existing ones. This is very useful: you get a type that has the same properties as some existing type but is treated as a distinct type in the interest of strong typing

type Social_Security_Number is new Integer range 0 .. 999_99_9999;

- Social_Security_Number *derived type*; its *parent type* is Integer
- you can refine the valid range when defining a derived scalar type (such as integer, floating-point and enumeration)

Subtypes

- types may be used in Ada to enforce constraints on the valid range of values
- enforce constraints on some values while staying within a single type
- a subtype does not introduce a new type
- subtypes of a type are statically compatible with each other, constraints are enforced at run time: if you violate a subtype constraint, an exception will be raised

```
subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
subtype Degree_Celsius is Float;
subtype Liquid_Water_Temperature is Degree_Celsius range 0.0 .. 100.0;
subtype Running_Water_Temperature is Liquid_Water_Temperature;
```

Subtypes as type aliases

- type aliases generate alternative names — *aliases* — for known types
- type aliases are sometimes called *type synonyms*
- using subtypes without new constraints

```
subtype Meters is Float;
subtype Miles is Float;
Dist_Imperial : Miles;
```

- we don't get all of the benefits of Ada's strong type checking
- recommendation is to use strong typing

```
Type X is new Y;
```

Array type declaration

- composite type / same typed elements
- define contiguous collections of elements that can be selected by indexing
- type My_Int is range 0 .. 1000;
- type Index is range 1 .. 5;
- type My_Int_Array is array (Index) of My_Int;

```
        --                                  ^ Type of elements
Arr : My_Int_Array := (2, 3, 5, 7, 11);
 --                 ^ Array literal, ^ aggregate in Ada
```

Arrays

- specify the index type for the array
- any discrete type is permitted to index an array, including Enum types
- querying an element of the array at a given index, same syntax as for function calls: array object followed by the index in parentheses
- initialize by aggregate
- bounds of an array can be any values
- bounds can vary, you should not assume / hard-code specific bounds when iterating / using arrays

Arrays

```
type Month_Duration is range 1 .. 31;
type Month is (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
type My_Int_Array is array (Month) of Month_Duration;
 --                                  ^ Can use an enumeration type as index
Tab : constant My_Int_Array := (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
--  Maps months to number of days (ignoring leap years)
--  constant declarations cannot be modified
Feb_Days : Month_Duration := Tab (Feb);   --  Nr of days in February
for M in Month loop
 Put_Line (Month'Image (M) & " has " & Month_Duration'Image
                                                (Tab (M))  & " days.");
```

Arrays - Indexing

- indexing operation is strongly typed
- wrong type to index the array - a compile-time error.
- arrays in Ada are bounds checked
- outside of the bounds - run-time error, no random access
- explicitly created an index type for the array or a range of values

```
type My_Array is array (1 .. 5) of Integer;
for I in 1 .. 5 loop  Put (Integer'Image (A(I)));
```

Arrays - Range attribute

for I in A'Range loop Put (Integer'Image (A(I)));

- array has an anonymous range for its bounds, since there is no name to refer to the range, Ada solves that via several attributes of array objects
- A'Range = A'First .. A'Last,
- A'First – index of the first element
- A'Last - index of the last element
- A'Length – nr of elements in an array
- "null array", which contains no elements / define an index range whose upper bound is less than the lower bound.

Arrays - Unconstrained arrays

- most powerful aspects of Ada's array
- fixed size: every instance of this type will have the same bounds and therefore the same number of elements and the same size
- bounds are not fixed: the bounds will need to be provided when creating instances of the type

```
type Days is (Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday, Sunday);

type Workload_Type is array (Days range <>) of Natural;

Workload : constant Workload_Type (Monday .. Friday)
:=(Friday => 7, others => 8); -- default value
```

Arrays

- Discrete_Type range <> ; -- serves as the type of the index, but different array instances may have different bounds from this type
- unconstrained array type, the bounds need to be provided when an instance is created
- aggregate syntax: associations by name, by giving the value of the index on the left side of an arrow
  association 1=>2 value 2 to element of index 1
- others => 8 --assign value 0 to every element that wasn't previously assigned in this aggregate

Arrays

- "box" notation: (<>) what is expected here can be anything
- unconstrained arrays in Ada, can pass them as parameters to subprograms or return them from functions, and they implicitly contain their bounds as part of their value, useless to pass the bounds or length
- different instances of the same unconstrained array type can have different bounds, a specific instance has the same bounds throughout its lifetime

Predefined array type: String

```
type String is array (Positive range <>) of Character;
A : String (1 .. 11) := "Hello World";
B : String (1 .. 11) := ('H', 'e', 'l', 'l', 'o', ' ','W', 'o', 'r', 'l', 'd');
for I in reverse A'Range Put (A (I));
```

- bounds can be omitted when creating an instance of an unconstrained array type if you supply an initialization
- bounds can be deduced from the initialization expression

```
Message : constant String := "dlroW olleH";
```

Arrays

- bounds *have* to be known when instances are created
- can change the values of elements in an array, you cannot change the array's bounds

```
declare
A : String := "Hello";
begin A := "World"; -- OK: Same size
A := "Hello World"; -- Not OK: Different size
end;
```

- violation will generally result in a run-time error
- *indefinite subtype*, which means that the subtype size is not known at compile time, but is dynamically computed

Arrays

- the return type of a function can be any type; a function can return a value whose size is unknown at compile time

function Get_Day_Name (Day : Days := Monday) return String;

- an array of Strings, the String subtype used as component will need to have a fixed size

type Days is (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

subtype Day_Name is String (1 .. 2); -- Subtype of string with known size

type Days_Name_Type is array (Days) of Day_Name;

Names : constant Days_Name_Type := ("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su");

Arrays slices

- take and use a slice of an array (a contiguous sequence of elements) as a name or a value
- A slice of an array is of the same type as the array, but has a different subtype, constrained by the bounds of the slice

```
Buf : String := "Hello ...";
Full_Name : String := "John Smith";
Buf (7 .. 9) := "Bob";
Put_Line (Buf);  --  Prints "Hello Bob"
Put_Line ("Hi " & Full_Name (1 .. 4)); --  Prints "Hi John"
```

Arrays

- objects can be renamed by using the renames
- allows for creating alternative names for these objects
- instead of explicitly using indices every time we're accessing certain positions of the array, we can create shorter names for these positions by renaming them

```
package Measurements is
subtype Degree_Celsius is Float;
Current_Temperature : Degree_Celsius;
end Measurements;
subtype Degrees is Measurements.Degree_Celsius;
T : Degrees renames Measurements.Current_Temperature;
Put_Line (Degrees'Image (T));
```

Records

- Records allow composing a value out of instances of other types
- each of the instances will be given a name
- the pair consisting of a name and an instance of a specific type is called a field, or a component
- record components can have default values
- the value can be any expression of the component type, and may be run-time computable

```
type Date is record Day : Integer range 1 .. 31; Month :
Month_Type := January; -- This component has a default
value
Year : Integer range 1 .. 3000 := 2012; -- Default value end
record;
```

Records - Aggregates

- a convenient notation for expressing values
- called aggregate notation, the literals called aggregates
- used to initalize records
- an aggregate is a list of values separated by commas and enclosed in parentheses
- allowed in any context where a value of the record is expected
- values for the components can be specified positionally or by name
- mixture of positional and named values is permitted, but cannot use a positional notation after a named one

```
Ada_Birthday : Date := (10, December, 1815); Leap_Day_2020 :
Date := (Day => 29, Month => February, Year => 2020);
```

Records - component selection

- to access components of a record instance, we use an operation that is called component selection

```
type Date is record
Day   : Integer range 1 .. 31;
Month : Month_Type;
Year  : Integer range 1 .. 3000 := 2032;
end record;
Some_Day : Date := (1, January, 2000);
Some_Day.Year := 2020;
Put_Line ("Day:" & Integer'Image (Some_Day.Day) & ", Month:
" & Month_Type'Image (Some_Day.Month) & ", Year:" &
Integer'Image (Some_Day.Year));
```