Once features
Once routines

If we have a call

\[ x.r \]

and \( x \) is not Void, the next thing to do is to execute the routine body of \( r \).

Right?

Wrong.

The routine might be a once routine.
Once routines

A routine body may start not only with do

\[
\text{\texttt{r is}} \\
\text{\texttt{do}} \\
\text{\texttt{end}} \\
\text{... Instructions ...}
\]

But also with the keyword \texttt{once}

\[
\text{\texttt{r is}} \\
\text{\texttt{once}} \\
\text{\texttt{end}} \\
\text{... Instructions ...}
\]

Where once is possibly followed by one or more “once keys”

\[
\text{\texttt{once ("THREAD"}}}.
\]
Once routines

In the basic case, when you write

\[ r \text{ is} \]
\[ \text{once} \]
\[ \text{... Instructions ...} \]
\[ \text{end} \]

without once keys, then Instructions will be executed at most once in the entire system execution.

The first time someone calls the routine, its body will be executed.

Any subsequent call will not execute the routine body.
Once functions

If the routine is a function

\[ r: \text{STRING} \text{ is} \]
\[ \text{once} \]
\[ \ldots \text{Instructions} \ldots \]
\[ \text{end} \]

The first time someone calls the function, its body will be executed and it will return its result normally.

Any subsequent call will not cause the execution of the routine body.

It will return immediately to the caller, giving as result the value computed by the first call.
To make sure that a library works on a properly initialized setup, write the initialization procedure as a Once and include a call to it at the beginning of every externally callable routine of the library.

class LIBRARY

feature {NONE}
  init is
  once
    ... Setup library ...
  end

feature --externally callable routines
  r is
    do
      init
      ...
    end
Once Uses: Shared Objects

To let various components of a system share an object, represent it as a once function that creates the object.

```plaintext
shared_object: SOME_REFERENCE_TYPE
    -- A single object useful to several clients
    once
    ... ; create Result
    end
```

Clients will “call” that function and in any case but the first, such a call returns a reference to the object created the first time around.
Predefined Once Keys

`once ("OBJECT")`  -- Once for each instance

`once ("THREAD")`  -- Once per execution of a thread  
                      -- (default)
Once Tuning

For more flexibility you can define your own once keys.

```python
once ("MY_KEY")
```

Lets you refresh a once key: The next call to a once routine that lists it as one of its once keys will execute its body.

To refresh once keys class ANY has a feature `onces` which you can use for calls like:

```python
onces.refresh("MY_KEY")
onces.refresh_some(["MY_KEY","OTHER_KEY"])
onces.refresh_all
onces.refresh_all_except(["MY_KEY","OTHER_KEY"])
onces.nonfresh_keys
```
Definition: Freshness of a once routine call

During execution, a call whose feature is a once routine $r$ is fresh if and only if every feature call started so far satisfies any of the following conditions:

1. It did not use $r$ as dynamic feature.

2. It was in a different thread and $r$ has the once key “THREAD” or no once key.

3. Its target was not the current object and $r$ has the once key “OBJECT”.

4. After it was started, a call was executed to one of the refreshing features of onces from ANY, including among the keys to be refreshed at least one of the once keys of $r$. 
A routine $r$ is fresh if:

- It hasn’t been called at all

- It has been called on different objects, and is declared once (“OBJECT”)

- It’s declared once (“MY_KEY”) and there has been, since the last applicable execution of $r$, a call `onces.refresh (“MY_KEY”)`
An applicable call makes r **unfresh** again, since the conditions have to apply to every call started so far.

```plaintext
feature
  some_routine
    once ("OBJECT")
    ...
  end
other_routine
do
  some_routine
end
  unfresh afterwards
```
Once Routine Semantics

Latest applicable target and result of a non-fresh call

The latest applicable target of a non-fresh call to a once routine \( df \) to a target object \( O \) is the last value to which it was attached in the call to \( df \) most recently started on:

1. If \( df \) has the once key "OBJECT" : \( O \).
2. Otherwise, if \( df \) has the once key "THREAD" or no once key: any target in the current thread.
3. Otherwise: any target in any thread.

If \( df \) is a function, the latest applicable result of the call is the last value returned by a fresh call using as target object its latest applicable target.
Once Routine Execution Semantics

The effect of executing a once routine \texttt{df} on a target object \texttt{O} is:

1. If the call is \textbf{fresh}: that of a non-once call made of the same elements, as determined by Non-once Routine Execution Semantics.

2. If the call is \textbf{not fresh} and the last execution of \texttt{df} on the latest applicable target triggered an exception: to trigger again an identical exception. The remaining cases do not then apply.

3. If the call is not fresh and \texttt{df} is a procedure: no further effect.

4. If the call is not fresh and \texttt{df} is a function: to attach the local variable \texttt{Result} to the latest applicable result of the call.
Once Routine Exceptions

“once a once exception, always a once exception!”

If the first call to a once routine yields an exception, then all subsequent calls for the same applicable target re-trigger the exception.

Clients that repeatedly ask for the same once routine, repeatedly get the same exception, telling them that the requested effect or value is impossible to provide.
Recursive Once Routines

- if a once routine is directly or indirectly recursive, its self-calls will not execute the body (in the absence of an intervening explicit refresh)

- for a function, they will return the Result as computed so far.

- With recursion a new call usually starts before the first one has terminated, so the result of the first call would not be a meaningful notion in this case.

- In this case the recursive call will return whatever value the first call has obtained so far for Result (starting with the default initialization)
Recursive Once Routines

- Recursive once functions are a bit bizarre, and of little apparent use, but no validity constraint disallows it.

```plaintext
recursive_once_routine (x: INTEGER): INTEGER is
  once
  Result := recursive_once_routine (x-1) + x
end

Call to recursive_once_routine would return the default initialization of INTEGER: 0.
```
Non-valid Once features

Once features are not allowed if

- the result type involves **Genericity**

  ```
  item: G
  once
  ...
  end
  ```

- used in a *creation procedure*

  A creation procedure must ensure the invariant, a non-fresh call to a creation procedure would not ensure the invariant anymore.
Conversion
Conformance determines when a type may be used in lieu of another.

Conformance relies on inheritance. The basic condition for $V$ to conform to $T$ is:

- The base class of $V$ must be a descendant of the base class of $T$. 
Convertibility completes the conformance mechanism.

Convertibility lets you perform assignment and argument passing in cases where conformance does not hold but you still want the operation to succeed.

Examples:

```plaintext
your_real := 10     -- your_real: REAL

routine.expecting_a_real (10)
```
Implicit Conversion

Most programming languages handle such cases through ad hoc rules applying to a fixed set of arithmetic types.

But there is no reason to stop at INTEGER and REAL.

With convertibility you can define a class COMPLEX that makes the assignment

c := 10

valid for c of type COMPLEX, with the effect of calling a conversion to assign to c the complex number [10.0, 0.0].
Implicit Conversion

Conversion is an abbreviation for an explicit form:

c := 10              -- Implicit

is a shorthand for

create c.from_integer (10)   -- Explicit

where class COMPLEX has a creation procedure
   from_integer that has been marked as a conversion
   procedure.
Conversion Basics

expanded class REAL_64 inherit ... create
  from_integer, from_real_32, ...

convert
  from_integer ({INTEGER}),
  from_real_32 ({REAL_32})

feature -- Initialization
  from_integer (n: INTEGER)
    -- Initialize by converting from n.
    do
      ... Conversion algorithm
    end
  ...
end
A procedure whose name appears in a Converters clause is a conversion procedure.

A type listed in a Converters clause is a conversion type.

convert
  from_integer ({INTEGER}),
  from_real_32 ({REAL_32})

conversion types INTEGER, REAL_32 convert to the current type REAL_64.
Conversion Procedure, Conversion Type

With conversion procedures you permit assignments and argument passing from the given conversion types to the current type.

This justifies mixed-type arithmetic expressions like:

your_real + your_integer

Since that notation is really a shortcut for a function call

your_real.plus (your_integer)
Conversion Queries

Conversion procedures allow conversions from $U$ (to $T$).

```plaintext
convert -- in class REAL_64
  from_integer ({INTEGER})
```

Conversion queries specify conversions to $T$ (from $U$).

```plaintext
convert -- in class INTEGER
  to_real_64: {INTEGER}
```

If you have access to both $T$ and $U$, you can use either mechanism.

But sometimes you only have access to $T$ or $U$. 
Conversion Queries: Example

In class STRING we can provide a conversion procedure as well as a function going the other way:

from_other (s: OTHER_STRING)
    -- Initialize from s.

to_other: OTHER_STRING
    -- Representation in “other” form of current string.
Conversion Queries: Example

Assume that `eiffel_routine` expects an Eiffel `STRING` and `external_routine` expects an `OTHER_STRING`.

We could either use explicit transformations:

```plaintext
eiffel_string: STRING
external_string: OTHER_STRING
...
eiffel_routine (create s.from_other (external_string))
external_routine (s.to_other)
```

or implicit transformations:

```plaintext
eiffel_routine (external_string)
external_routine (s)
```
Conversion Queries: Example

In the implicit case, if restricted to conversion procedures, you would need to add a conversion procedure to \texttt{OTHER\_STRING}.

This won't work if \texttt{OTHER\_STRING} is an external class over which you have no control.

You can work from the other side in such cases - by marking \texttt{to\_other} as a conversion query:

\begin{verbatim}
create
  from\_other
convert
  from\_other \{(\texttt{OTHER\_STRING})\},
  to\_other: \{(\texttt{OTHER\_STRING})\}
\end{verbatim}
Using Conversion Properly

You never **have** to include a conversion facility. You can either use the explicit form:

```create` target.conversion_procedure (source)`
```

Or make use of the conversion facility:

```target := source```

A general advice is:

A creation procedure should provide a conversion mechanism only if the associated operation does not entail any loss of information.

```real := integer` -- OK`
integer := real` -- Information loss`
integer := real.truncated` -- OK (explicit)```
Conversion Principles

- **Conversion Principle:**
  No type may both conform and convert to another.

- **Conversion Asymmetry Principle:**
  No type $T$ may convert to another through both a conversion procedure and conversion query.

- **Conversion Non-Transitivity Principle**
  That $V$ converts to $U$ and $U$ to $T$ does not imply that $V$ converts to $T$. 
Conversion Principle

When you read
\[ x := y \]
You see immediately which of convertibility or conformance applies:

- If \( y \) conforms to \( x \), because \( TY \) and \( TX \) are the same or there is direct inheritance between their base classes, there is no conversion involved.
- If the class texts specify that \( TY \) converts to \( TX \), the attachment will involve a conversion.

If an attachment involves a conversion, it’s always because the types don’t conform.
Conversion Asymmetry Principle

Ensures that a conversion through a procedure cannot compete with a conversion through a query.

\[
\text{from\_string (s: STRING) } \quad -- \text{ in class OTHER\_STRING}
\]

\[
\text{to\_other: OTHER\_STRING } \quad -- \text{ in class STRING}
\]

\[
\text{my\_other\_string := my\_string}
\]

Which conversion feature will be used?
Conversion Non-Transitivity Principle

Convertibility is always the result of an explicit convert clause.

Conversion involves a transformation of values, which should always be explicit.

With transitivity, additional transformations would occur behind the scene.

Therefore convertibility is non-transitive.
Explicit Conversion

Conversion usually happens implicitly as a result of a reattachment:

```plaintext
var := exp -- var: T, exp:U
```

Additionally there is a library feature that explicitly implies a conversion:

```plaintext
{T} [exp]
```

If $U$ converts to $T$, this expression denotes the result of converting $exp$ to $T$, and is called an *explicit conversion*. 
Explicit Conversion

Not a language mechanism to achieve this:

The Kernel Library class \texttt{TYPE [G]} provides a function

\texttt{adapted alias "[]" (x: G): G}

which can be used for any type \texttt{T} and any expression \texttt{exp}

of a type \texttt{U} compatible with \texttt{T} to produce a \texttt{T} version of \texttt{exp}.

\texttt{\{T\} [exp]}

is equivalent to

\texttt{t_type.adapted (exp)} \quad \text{-- t_type: TYPE [T]}

Where \texttt{\{T\}} represents \texttt{TYPE [T]}
Explicit Conversion: Example

\{DATE\} [[20, “April”, 2007]]

The tuple is given as argument to the adapted function of \{DATE\} that turns it into an expression of type DATE.

This is permitted if class DATE specifies a conversion procedure from TUPLE [NATURAL, STRING, NATURAL] to DATE.
Explicit Conversion: Example

```python
compute_revenue ([1, "Januar", 2007], [31, "May", 2007])
```

Where `compute_revenue` is expecting two arguments of type `DATE` works also without specifying `{DATE}`.

The need for explicit conversion arises because conversion is not transitive.
Explicit Conversion: Example

process_date_string (s: STRING)
Assume that DATE converts to STRING and the conversion from tuple to DATE is still valid:

process_date_string ([1, “Januar”, 2007])
is invalid as it requires two conversions.

process_date_string ({DATE} [[1, “Januar”, 2007]])
is valid as it has only one implicit conversion.

It is possible to specify several conversions in this style:
{T} [{U} [of_type_V]]
Target Conversion

```plaintext
my_integer + my_real
my_integer.plus (my_real)
```

We don’t want to use feature `plus` of class `INTEGER`, but convert `my_integer` to `REAL` and then use feature `plus` of class `REAL`.

The target conversion mechanism makes this possible. It is supported syntactically by the optional convert mark of an Alias for an infix feature:

```plaintext
plus alias “+” convert (other: INTEGER): INTEGER
```
Target Conversion

plus alias “+” convert (other: INTEGER): INTEGER

The convert mark means:
If the type of the argument is not the normally expected argument type, but one to which the current type converts, such as REAL:

1. Convert the target to the applicable conversion type

2. Ignore the implementation here (integer addition)

3. Instead call the function with the same name from the applicable conversion type (here REAL)
Purpose to support traditional conventions of mathematical notation, in particular mixed-type arithmetic expressions.

Given that this is the usual expectation for numerical computation, four approaches are possible in an object-oriented language:
1. **Ignore the issue:**
   Treat arithmetic types as completely different from O-O types. (Done in Java, C++)

2. **Insist on purity:**
   Mixed-type expressions are wrong and one should use explicit conversion. (Programmers expect to be able to use mixed-type expressions)

3. **Provide special rules for basic types:**
   Keep integers, reals and such within the O-O type system. Introduce special cases in the conformance and reattachment rules. (Used in Eiffel 3, couldn’t handle expressions like my_integer + my_real)

4. **Provide a general conversion mechanism:**
   As seen here.
End of Lecture 11