Routine: algorithm abstraction

To abstract is to capture the essence behind the details and the specifics

Implies giving a name to the result

In programming:
- Data abstraction: class
- Algorithm (operational) abstraction: routine

A routine is one of the two kinds of feature...

... the other is attribute

We have encountered lots of routines already, without the name.
A routine

```pseudocode
r(arg: TYPE; ...) is
  -- Header comment
  require
    ... Precondition (boolean expression) ...
  do
    ... Body (instructions) ...
  ensure
    Postcondition (boolean expression)
end
```

Uses of routines

Bottom-up: capture existing algorithm, possibly for reuse

Top-down: placeholder routines — attractive alternative to pseudocode.

```
build_a_line is
  -- Build imaginary line
  do
    Paris.display
    Menu_highlight
    create_fancy_line
  end
create_fancy_line is
  -- Create line and fill stations
  do
    -- To be completed
    BM: 29 Nov 05
  end
```

Two kinds of routine

Procedure: doesn't return a result
  ➢ Yields a command
  ➢ Calls are instructions

Function: returns a result

```pseudocode
f(arg: TYPE; ...) RESULT_TYPE is
  ... (The rest as before) ...
  ➢ Yields a query
  ➢ Calls are expressions
```

---

Page 2
Features: the full story

A class is characterized by its features
Each feature is an operation on the corresponding objects: query or command
Features are grouped into categories for readability
Class clauses:
- Indexing
- Inheritance
- Creation
- Feature (any number)
- Invariant

Anatomy of a class:

Features: the full story

Client view (specification)
- Command
- Procedure
- No result
- Returns result
- Query

Internal view (implementation)
- Routine
- Compute
- Memory
- Function
- Attribute

Uniform access principle

It doesn’t matter to the client whether you look up or compute

A call such as

`ibm_company.yearly_average`

could use an attribute or a function
An object has an **interface**

An object has an **implementation**

Information hiding
**Uniform Access Principle**

Expressed more technically:

Features should be accessible to clients the same way whether implemented by storage or by computation.

---

**Uniform Access: an example**

\[
\text{balance} = \text{list\_of\_deposits\_total} - \text{list\_of\_withdrawals\_total}
\]

- **(A1)**
  - list of deposits
  - list of withdrawals
  - balance

- **(A2)**
  - list of deposits
  - list of withdrawals

---

**Uniform Access Principle**

Features should be accessible to clients the same way whether implemented by storage or by computation.
What clients may do

```plaintext
class METRO_STATION feature
  x, y: REAL      -- Coordinates of metro station
  size: REAL     -- Size of bounding square
  upper_left: POSITION
                  -- Upper-left position of bounding square
  adjust_positions is
    do
      upper_left x = 3
    end
end
```

What clients may not do

```plaintext
class METRO_STATION feature

  adjust_positions is
    do
      upper_left x = 3
    end
end
```

Use procedures:

```plaintext
upper_left: set(3, upper_left, y)
upper_left: set_x(3)

upper_left: move(3, h)
```
Possible client privileges

If class \( A \) has an attribute \( att: \text{SOME\_TYPE} \), what may a client class \( C \) with
\[ a: A \]
do with \( a \cdot att \)? The attribute may be:

- Secret
- Read-only
- Read
- restricted write
- Full write

Example: modify \( x \) with move in \( \text{POINT} \)
Modify through "set..." procedure

Abstraction and client privileges

If class \( A \) has an attribute \( att: \text{SOME\_TYPE} \), what may a client class \( C \) with
\[ a: A \]
do with \( a \cdot att \)?

Read access if attribute is exported
- \( a\cdot att \) is an expression.
- An assignment \( a\cdot att \rightarrow y \) would be syntactically illegal.

(It would assign to an expression, like \( x \rightarrow y \rightarrow k \).)
### Applying abstraction principles

Beyond read access: full or restricted write, through exported procedures.

Full write privileges: `set_attribute` procedure, e.g.

```plaintext
set_temperature (u: REAL) is
  -- Set temperature value to u
  do
    temperature := u
  ensure
    temperature_set: temperature = u
end
```

Client will use e.g. `x.set_temperature (21.5)`.

### Other uses of a setter procedure

```plaintext
set_temperature (u: REAL) is
  -- Set temperature value to u
  require
    not_under_minimum: u >= -273
    not_above_maximum: u <= 2000
  do
    temperature := u
    update_database
  ensure
    temperature_set: temperature = u
end
```

### Having it both ways

Make it possible to call a setter procedure

```plaintext
temperature: REAL assign set_temperature
```

Then the syntax

```plaintext
x.temperature := 21.5
```

is accepted as a shorthand for `x.set_temperature (21.5)`

Retains contracts etc.
Information hiding

- **class A**
  - **feature f**
  - **feature g**
  - **feature (NONE)**
    - **h**
  - **feature (R, A)**
    - **j**, **k**, **l**
  - **feature (A, B, C)**
    - **m**, **n**
  - **end**

Status of calls in a client with **A**: A:

- `a1, f, a1, g` valid in any client
- `a1, h` invalid everywhere (including in A's own text)
- `a1, j` valid only in B, C and their descendants (not valid in A)
- `a1, m` valid in B, C and their descendants, as well as in A and its descendants

An example of selective export

**LINKABLE** exports its features to **LINKED_LIST**

Does not export them to the rest of the world

Clients of **LINKED_LIST** don't need to know about **LINKABLE** cells.

```
count 2

tip, element
```

```
Haldenap
    left  right
    item
```
```
Central
    left  right
    item
```
```
Target
    left  right
    item
```

Exporting selectively

```
class LINKABLE[G]
feature (LINKED_LIST)
  put_right(...) is do ... end
  right: G is do ... end
  ...
end
```
Information hiding only applies to use by clients, using dot notation or infix notation, as with .f. (Qualified calls).

Unqualified calls (within class) not subject to information hiding:

```plaintext
class A feature (NONE)
    h is ... do ... end
feature
    f is ...
    do ...
    end
end
```

End of lecture 10