Lecture 13: Introduction to inheritance and genericity
On the menu for today (& next time)

Two fundamental mechanisms for expressiveness and reliability:
- **Genericity**
- **Inheritance**

with associated (just as important!) notions:
- **Static typing**
- **Polymorphism**
- **Dynamic binding**
Extending the basic notion of class

Inheritance

Abstraction

SET_OF_CARS

LIST_OF_CARS

LIST_OF_CITIES

LIST_OF_CARS

LINKED_LIST_OF_CARS

Genericity

Type parameterization

LIST_OF_PERSONS

LIST_OF_CARS

Type parameterization

Specialization

Introduction to Programming, lecture 13: Inheritance & Genericity
Reminder: the dual nature of classes

A class is a module
A class is a type*

As a module, a class:
- Groups a set of related services
- Enforces information hiding (not all services are visible from the outside)
- Has clients (the modules that use it) and suppliers (the modules it uses)

As a type, a class:
- Denotes possible run-time values (objects & references), the instances of the type
- Can be used for declarations of entities (representing such values)

*Or a type template (see genericity)
Inheritance basics

Principle:

Describe a new class as extension or specialization of an existing class
(or several with *multiple* inheritance)

If $B$ inherits from $A$:

- **As modules**: all the services of $A$ are available in $B$
  (possibly with a different implementation).

- **As types**: whenever an instance of $A$ is required, an instance of $B$
  will be acceptable
  ("is-a" relationship)
Terminology

Parent, Heir

Ancestor, Descendant
The ancestors of $B$ are $B$ itself and the ancestors of its parents.

Proper ancestor, Proper descendant

Direct instance, Instance
The instances of $A$ are the direct instances of its descendants.

(Other terminology: subclass, superclass, base class)
If \( B \) inherits from \( A \) (by listing \( A \) in its `inherit` clause), \( B \) is an **heir** of \( A \) and \( A \) is a **parent** of \( B \).

The **descendants** of a class are the class itself and (recursively) the descendants of its heirs; **proper descendants** exclude the class itself.

“Ancestor” and “proper ancestor” are the reverse notions.
Example hierarchy

* deferred
+ effective
++ redefined

MOVING

load
position
update_coordinates
move

VEHICLE

TAXI

EVENT_TAXI

take*
take+

LINE_VEHICLE

TRAM

BUS

update_coordinates++
move++
Features

Feature (for the examples):

- **take** *(from_location, to_location: COORDINATE)*
  -- Bring passengers from *from_location*
  -- to *to_location*.

- **busy**: BOOLEAN
  -- Is taxi busy?

- **load** *(q: INTEGER)*
  -- Load *q* passengers.

- **position**: TRAFFIC_COORDINATE
  -- Current position on map

From class:

- **EVENT_TAXI**
- **TAXI**
- **VEHICLE**
- **MOVING**
Inheriting features

```plaintext
class EVENT_TAXI
  inherit TAXI
  feature [... Rest of class ...]
end
defered class TAXI
  inherit VEHICLE
  feature [... Rest of class ...]
end
defered class VEHICLE
  inherit MOVING
  feature [... Rest of class ...]
end
```

All features of TAXI are applicable to instances of EVENT_TAXI

All features of VEHICLE are applicable to instances of TAXI

All features of MOVING are applicable to instances of VEHICLE
Inheritance hierarchy

** MOVING **

- position
- update_coordinates
- move

** VEHICLE **

- load

** TAXI **

- busy
- take*

** LINE_VEHICLE **

- update_coordinates++
- move++

** EVENT_TAXI **

- take+

** TRAM **

** BUS **

* deferred
+ effective
++ redefined
Inheritted features

\[ m: MOVING, v: VEHICLE, t: TAXI, e: EVENT_TAXI \]

\[ v \cdot load (...) \]
\[ e \cdot take (...) \]
\[ m \cdot position \quad \text{-- An expression} \]
\[ t \cdot busy \quad \text{-- An expression} \]
\[ e \cdot load (...) \]
\[ e \cdot take (...) \]
\[ e \cdot position \quad \text{-- An expression} \]
\[ e \cdot busy \quad \text{-- An expression} \]
A "feature of a class" is one of:

- An **inherited** feature if it is a feature of one of the parents of the class.
- An **immediate** feature if it is declared in the class, and not inherited. In this case the class is said to **introduce** the feature.
Polymorphic assignment

\( v : \text{VEHICLE} \)
\( \textit{cab} : \text{EVENT\_TAXI} \)

\( v := \text{cab} \)

A proper descendant type
Assignments

So far (no polymorphic assignments):

\[ \text{target} := \text{expression} \]

where \text{expression} is of the \text{same type as target}.

With polymorphism:

\[ \text{target} := \text{expression} \]

where the type of \text{expression} is a \text{descendant} of the type of \text{target}.
Polymorphism is also for argument passing

\[ \text{register\_trip}(v: \text{VEHICLE}) \]

\[ \text{register\_trip}(\text{cab}) \]

A proper descendant type
Definitions: Polymorphism

An attachment (assignment or argument passing) is **polymorphic** if its target variable and source expression have different types.

An entity or expression is **polymorphic** if it may at runtime — as a result of polymorphic attachments — become attached to objects of different types.

**Polymorphism** is the existence of these possibilities.
Definitions (Static and dynamic type)

The **static type** of an entity is the type used in its declaration in the corresponding class text.

If the value of the entity, during a particular execution, is attached to an object, the type of that object is the entity’s **dynamic type**.
Static and dynamic type

\( v: VEHICLE \)
\( cab: EVENT\_TAXI \)

\( v := cab \)

Static type: \( VEHICLE \)
Dynamic type after this assignment: \( EVENT\_TAXI \)

For a polymorphic attachment to be valid, the type of the source must conform to the type of the target.
Definition: Conformance

A (non-generic) reference type $U$ **conforms** to a reference type $T$ if $U$ is a descendant of $T$.

An expanded type conforms only to itself.
Another example hierarchy

FIGURE

OPEN FIGURE

SEGMENT

POLYLINE

POLYGON

CLOSED FIGURE

ELLIPSE

TRIANGLE

RECTANGLE

CIRCLE

* deferred
+ effective
++ redefine
Redefinition 1: polygons

class POLYGON inherit CLOSED FIGURE
create
  make
feature
  vertex: ARRAY [POINT]
  vertex_count: INTEGER
  perimeter: REAL is
    -- Perimeter length
    do
      from ... until ... loop
        Result := Result + vertex[i] \cdot distance(vertex[i + 1])
      end
    end
  invariant
    vertex_count >= 3
    vertex_count = vertex.count
end
Redefinition 2: rectangles

class RECTANGLE inherit POLYGON
   redefine perimeter
end
create
make
feature
diagonal, side1, side2: REAL
   perimeter: REAL is
      -- Perimeter length
      do Result := 2 * (side1 + side2) end
invariant
   vertex_count = 4
end
Assume:

\[ p: \text{POLYGON} \; ; \; r: \text{RECTANGLE} \; ; \; t: \text{TRIANGLE}; \]
\[ x: \text{REAL} \]

Permitted:

\[ x := p\. \text{perimeter} \]
\[ x := r\. \text{perimeter} \]
\[ x := r\. \text{diagonal} \]
\[ p := r \]

NOT permitted:

\[ x := p\. \text{diagonal} \quad \text{-- Even just after } p := r \]
Dynamic binding

What is the effect of the following (assuming `some_test` true)?

```plaintext
if some_test then
    p := r
else
    p := t
end
x := p.perimeter
```

**Redefinition:** A class may change an inherited feature, as with `POLYGON` redefining perimeter.

**Polymorphism:** `p` may have different forms at run-time.

**Dynamic binding:** Effect of `p.perimeter` depends on run-time form of `p`. 
Without dynamic binding!

```
display (f: FIGURE) is
    do
        if "f is a CIRCLE" then
            ...
        elseif "f is a POLYGON" then
            ...
        end
    end

and similarly for all other routines!

Tedious; must be changed whenever there's a new figure type.
```
With inheritance and associated techniques

With:

\[
\begin{align*}
f & : \text{FIGURE} \\
c & : \text{CIRCLE} \\
p & : \text{POLYGON}
\end{align*}
\]

and:

\[
\begin{align*}
\text{create } c & .\text{make } (...) \\
\text{create } p & .\text{make } (...)
\end{align*}
\]

Initialize:

\[
\begin{align*}
\text{if } ... & \text{ then } f := c \\
\text{else } & f := p \\
\text{end}
\end{align*}
\]

Then just use:

\[
\begin{align*}
f & .\text{move } (...) \\
f & .\text{rotate } (...) \\
f & .\text{display } (...) \\
& \text{-- and so on for every} \\
& \text{-- operation on } f !
\end{align*}
\]
Inheritance: summary 1

Type mechanism: lets you organize our data abstractions into taxonomies

Module mechanism: lets you build new classes as extensions of existing ones

Polymorphism: Flexibility *with* type safety

Dynamic binding: automatic adaptation of operation to target, for more modular software architectures
Redefinition

defered class MOVING feature
     origin: COORDINATE
destination: COORDINATE
position: COORDINATE
polycursor: ARRAYED_LIST_CURSOR [COORDINATE]
update_coordinates
     -- Update origin and destination.
     do
        [...]
        origin := destination
        polycursor.forth
        destination := polycursor.item
        [...]
     end
 [...]
end
Redefinition 2: LINE_VEHICLE

defered class LINE_VEHICLE inherit
  VEHICLE
  redefine update_coordinates end
feature
  linecursor: LINE_CURSOR
  update_coordinates is
    -- Update origin and destination.
    do
      [...]  
      origin := destination
      polycursor.forth
      if polycursor.after then
        linecursor.forth
        create polycursor.make (linecursor.item.polypoints)
        polycursor.start
      end
      destination := polycursor.item
    end
Dynamic binding

What is the effect of the following (assuming \texttt{some\_test} true)?
\texttt{m: MOVING, l: LINE\_VEHICLE, t: TAXI}

\begin{verbatim}
if some\_test then
  m := l
else
  m := t
end
\end{verbatim}

\texttt{m.update\_coordinates}

\textbf{Redefinition}: A class may change an inherited feature, as with \texttt{LINE\_VEHICLE} redefining \texttt{update\_coordinates}.

\textbf{Polymorphism}: \texttt{m} may have different forms at run-time.

\textbf{Dynamic binding}: Effect of \texttt{m.update\_coordinates} depends on run-time form of \texttt{m}.
There are multiple versions of `take`.

Inheritance diagram:

- `TAXI` inherits from:
  - `EVENT_TAXI`
  - `DISPATCHER_TAXI`

- Methods:
  - `busy`
  - `take*`
  - `take`
Definitions (Dynamic binding)

**Dynamic binding** (a semantic rule) is the property that any execution of a feature call will use the version of the feature best adapted to the type of the target object.
Extending the basic notion of class

- Abstraction
- Inheritance
- Genericity
- Specialization

**List of Classes:**
- LIST_OF_CARS
- SET_OF_CARS
- LIST_OF_CITIES
- LINKED_LIST_OF_CARS
- LIST_OF_PERSONS

Type parameterization:
- LIST_OF_CARS
- LIST_OF_CITIES
- LIST_OF_PERSONS

Introduction to Programming, lecture 13: Inheritance & Genericity
Extending the basic notion of class

Inheritance

LIST_OF_CARS

SET_OF_CARS

SET_OF_PERSONS

LIST_OF_CITIES

LIST_OF_CARS

LIST_OF_PERSONS

LINKED_LIST_OF_CITIES

LINKED_LIST_OF_CARS

Genericity
Genericity

**Unconstrained**

\[
\text{LIST} [G]
\]

- e.g. \(\text{LIST} [\text{INTEGER}]\), \(\text{LIST} [\text{PERSON}]\)

**Constrained**

\[
\text{HASH\_TABLE} [G \rightarrow \text{HASHABLE}]
\]

\[
\text{VECTOR} [G \rightarrow \text{NUMERIC}]
\]
Genericity: Ensuring type safety

How can we define consistent “container” data structures, e.g. list of accounts, list of points?

Dubious use of a container data structure:

```plaintext
\text{c} : \text{CITY} ; \text{p} : \text{PERSON}
\text{cities} : \text{LIST} ...
\text{people} : \text{LIST} ...

people.extend ( )
cities.extend ( )

\text{c} := \text{cities.last}
```

c. \text{trans.force} (\text{Line8, Line8.name})
Possible approaches

1. Duplicate code, manually or with help of macro processor.

2. Wait until run time; if types don’t match, trigger a run-time failure. (Smalltalk)

3. Convert ("cast") all values to a universal type, such as "pointer to void" in C.

4. Parameterize the class, giving an explicit name $G$ to the type of container elements. This is the Eiffel approach, now also found in Java, .NET and others.
A generic class

class \textit{LIST}[G] \textit{ feature}
   \textit{extend}(x: G) \textit{ is } ...
   \textit{last}: G \textit{ is } ...
end

To use the class: obtain a \textit{generic derivation}, e.g.

\textit{cities}: \textit{LIST}[\textit{CITY}]

Introduction to Programming, lecture 13: Inheritance & Genericity
Using generic derivations

\texttt{cities : LIST[CITY]}
\texttt{people : LIST[PERSON]}
\texttt{c : CITY}
\texttt{p : PERSON}

... 

\texttt{cities.extend (c)  
people.extend (p)}

\texttt{c := cities.last}
\texttt{c.lines.force (Line8, Line8.name)}

\textbf{STATIC TYPING:}

The compiler will reject:

- \texttt{people.extend (c)}
- \texttt{cities.extend (p)}
Static typing

**Type-safe call** (during execution):

A feature call $x.f$ such that the object attached to $x$ has a feature corresponding to $f$.

[Generalizes to calls with arguments, $x.f(a, b)$ ]

**Static type checker:**

A program-processing tool (such as a compiler) that guarantees, for any program it accepts, that any call in any execution will be *type-safe*.

**Statically typed language:**

A programming language for which it is possible to write a *static type checker.*
Using genericity

LIST [CITY]
LIST [LIST [CITY]]
...

A type is no longer exactly the same thing as a class!

(But every type remains based on a class.)
Polymorphic data structures

fleet: LIST [VEHICLE]
v: VEHICLE

extend (v: G)

-- Add a new occurrence of v.

... fleet.extend (v) fleet.extend (cab)
Definition (Polymorphism, adapted)

An attachment (assignment or argument passing) is polymorphic if its target entity and source expression have different types.

An entity or expression is polymorphic if – as a result of polymorphic attachments – it may at runtime become attached to objects of different types.

A container data structure is polymorphic if it may contain references to objects of different types.

Polymorphism is the existence of these possibilities.
Definitions (Conformance, adapted)

A reference type $U$ **conforms** to a reference type $T$ if either:

- They have no generic parameters, and $U$ is a descendant of $T$.

- They are both generic derivations with the same number of actual generic parameters, the base class of $U$ is a descendant of the base class of $T$, and every actual parameter of $U$ (recursively) conforms to the corresponding actual parameter of $T$.

An expanded type conforms only to itself.
What we have seen

The basics of fundamental O-O mechanisms:

- Inheritance
- Polymorphism
- Dynamic binding
- Static typing
- Genericity
End of lecture 13