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
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)
**Terminology**

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**

![Example hierarchy diagram](image)

---

**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

Inheritance hierarchy

Inherited features

---

Inheriting features

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

---

Inherited features

m: MOVING, v: VEHICLE, t: TAXI, e: EVENT_TAXI

v-load (...)  e-take (...)  m-position -- An expression  t-busy -- An expression

e-load (...)  e-take (...)  e-position -- An expression  e-busy -- An expression

---
Definitions: kinds of features

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

```plaintext
v: VEHICLE
cab: EVENT_TAXI
v := cab
```

Assignments

So far (no polymorphic assignments):

```
target := expression
```

where expression is of the same type as target.

With polymorphism:

```
target := expression
```

where the type of expression is a descendant of the type of target.
Polymorphism is also for argument passing

```
register_trip(\* VEHICLE)
```

```
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**.
Introduction to Programming, lecture 13: Inheritance & Genericity

Static and dynamic type

```
v := cab
```

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

![Diagram showing an example hierarchy of types including `FIGURE`, `SEGMENT`, `POLYLINE`, `POLYGON`, `ELLIPSE`, `CIRCLE`, `RECTANGLE`, `TRIANGLE`, `SQUARE`, and various methods like `center`, `display`, `rotate`, `perimeter`, `side1`, `side2`, `perimeter*`, `perimeter++`, `perimeter***`, `method1`, and `method2`.](image)
Redefinition 1: polygons

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

Redefinition 2: rectangles

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

Inheritance, typing and polymorphism

Assume:

- p: POLYGON
- r: RECTANGLE
- t: TRIANGLE
- x: REAL

Permitted:

- x := p.perimeter
- x := r.perimeter
- x := r.diagonal
- p := r

NOT permitted:

- x := p.diagonal -- Even just after p := r!
- r := p
Dynamic binding

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

```java
if some_test then
    p := r
else
    p := t
end
```

```java
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!

```java
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:

```java
f : FIGURE
c : CIRCLE
p : POLYGON
```

and:

```java
create c.make (...) create p.make (...)```

Initialize:

```java
if ... then
    f := c
else
    f := p
end```

Then just use:

```java
f.move (...) f.rotate (...) f.display (...) -- and so on for every -- operation on f!
```
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

defined class MOVING

define

define

feature

origin: COORDINATE

destination: COORDINATE

polycursor: ARRAYED_LIST_CURSOR[COORDINATE]

update_coordinates

do

[...]

origin := destination

polycursor.forth

[...]

end

end

Redefinition 2: LINE_VEHICLE

defined class LINE_VEHICLE inherit

feature

linecursor: LINE_CURSOR

update_coordinates is

-- Update origin and destination.

do

[...]

origin := destination

polycursor.forth

if polycursor.after

linecursor.forth

create polycursor.make_linecursor(linecursor.item,polypoints)

polyoptimizer.start

end

[...]

destination := polycursor.item

end

end
**Dynamic binding**

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

```
  m MOVING, l LINE_VEHICLE, t TAXI
  if some_test then
    m := l
  else
    m := t
  end
  m.update_coordinates
```

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

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

**Dynamic binding:** Effect of `m.update_coordinates` depends on run-time form of `m`.

---

**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.
Introduction to Programming, lecture 13: Inheritance & Genericity

Extending the basic notion of class

- Abstraction
- Inheritance
- Generality

Type parameterization

LIST_OF_CARS

SET_OF_CARS

LIST_OF_CITIES

LIST_OF_PERSONS

LINKED_LIST_OF_CARS

Genericity

Unconstrained

LIST [G]
e.g. LIST [INTEGER], LIST [PERSON]

Constrained

HASH_TABLE [G → HASHABLE]
VECTOR [G → 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:

```
c : CITY; p : PERSON

cities : LIST...
people : LIST...
```

```
people.extend( )
cities.extend( )
```

```
c := cities.last
```

```
c. trans.force (Line8, Line8.name)
```

What if wrong?

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 LIST[\( G \)] feature
extend(x : \( G \)) is ...
last : \( G \) is ...
end
```

To use the class: obtain a generic derivation, e.g.

```
cities : LIST[CITY]
```

Formal generic parameter

Actual generic parameter
Using generic derivations

```
cities : LIST[CITY]
people : LIST[PERSON]
c : CITY
p : PERSON

...```

```
cities.extend (c)
people.extend (p)
```

```
c := cities.last
c.lines.force (Line8, Line8.name)
```

STATIC TYPING:

The compiler will reject:
- `people.extend (c)`
- `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

```plaintext
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