Inheritance basics

Principle:

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

If \( B \) inherits from \( A \):

- Module viewpoint: all the services of \( A \) are available in \( B \) (possibly with a different implementation).
- Type viewpoint: 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)

Example hierarchy

Redefinition 1: polygons

```plaintext
class POLYGON inherits CLOSED FIGURE
create
take
feature
  vertex : ARRAY [POINT]
  vertex_count : INTEGER
  perimeter : REAL
    do
      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
    -- Perimeter length
    do Result := 2 ∗ (side1 + side2) end
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)?

  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 inheritance!

```plaintext
display(f: FIGURE) is
do
  if "f is a CIRCLE" then
  ... 
  elseif "f is a POLYGON" then 
  ... 
end
and similarly for all other routines!
Tedious; must be changed whenever there's a new figure type.
```

With inheritance!

```plaintext
With:
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
**Genericity**

*Unconstrained*

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

*Constrained*

- HASH_TABLE [G \rightarrow HASHABLE]
- VECTOR [G \rightarrow NUMERIC]

**Extending the basic notion of class**

- SET OF CARS
- LIST OF CARS
- LINKED LIST OF CARS
- LIST OF CITIES
- LIST OF PERSONS
- SET OF CARS
- SET OF CITIES
- SET OF PERSONS
- LINKED LIST OF CITIES
- LINKED LIST OF PERSONS
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
c : CITY; p : PERSON

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

----------------------------------------------
people.extend ( )
cities.extend ( )

c := cities.last

c.add_tram_line (Line8)
```

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

```plaintext
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.add_tram_line (Line8)
```

**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**.

---

The static and the dynamic

For a feature call \(x.f\):

**Static typing:**

There is at least one feature \(f\) applicable to \(x\)

**Dynamic binding:**

If more than one possible feature, execution will select the right feature
Using genericity

\[
\text{LIST [CITY]}
\text{LIST [LIST [CITY]]}
\]

A type is no longer exactly the same thing as a class!
(But every type remains based on a class.)

What we have seen

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