Inheritance
&
Adaptation

Need for adaptation

- Suppose we have a class for which we do not have the program text
  - » **All we have is the interface**

Need for adaptation – 2

- Suppose we have a class for which we do not have the program text
  - » **All we have is the interface**

- We want to modify the class
  - » **How? it is closed**

Need for adaptation – 3

- Suppose we have a class for which we do not have the program text
  - » **All we have is the interface**
- We want to modify the class
  - » **How? it is closed**
- We need to be able to open the class for modification
  - » **to change features**
  - » **add new features**
  - » **remove features**

Open-Closed Principle

- Open – Available for extension – add new features

- Closed – Available for client use – stable in spite of extensions

In real projects
A module needs to be both open and closed!

Open-Closed Principle – 2

» **How is the open-closed principle implemented in OO languages?**

Open-Closed Principle – 3

- Inheritance
  - » **Allows us to re-open a class after it is closed**
  - » **It is the mechanism that makes the open-closed principle possible**

Open-Closed Principle – 4

- Inheritance
  - » **Allows us to re-open a class after it is closed**
  - » **It is the mechanism that makes the open-closed principle possible**

- In general, a child class inherits all the features from a parent class
  - » **Though OO languages allow us to modify the inherited features**

Invariant Inheritance Rule

The invariant property of a class is the Boolean and of the assertions appearing in its invariant clause, and of the invariant properties of its parents if any.

Creation Inheritance Rule

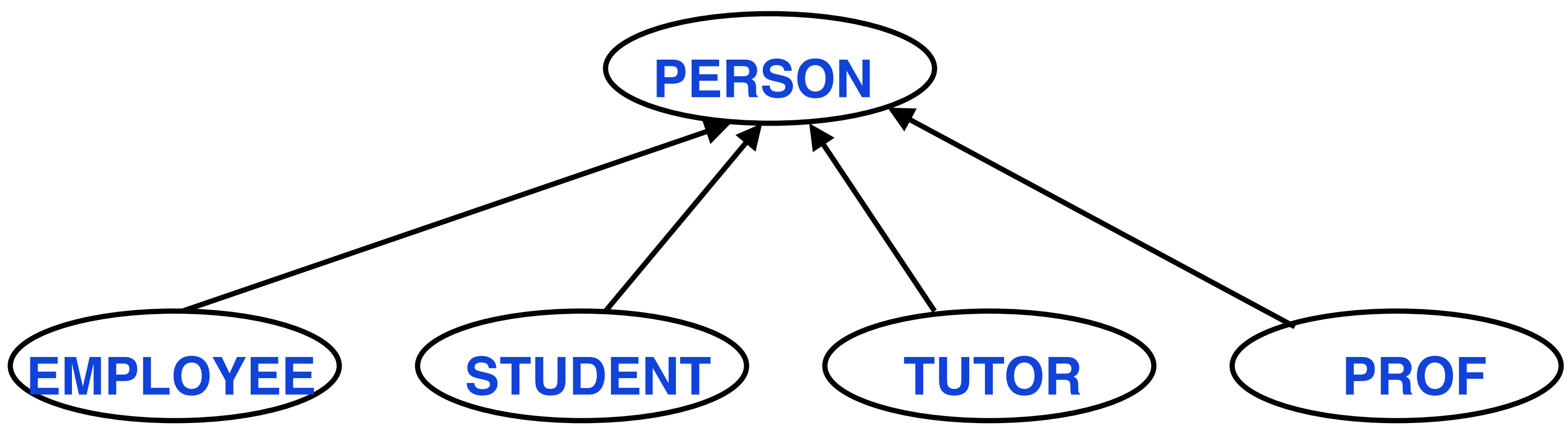
An inherited feature's creation status in a parent class (whether or not the feature is a creation method) has no bearing on its creation status in the child class.

Feature Adaptation

- Under inheritance a new class may share behaviour of a parent class, but may need to modify it

Feature Adaptation – 2

- Under inheritance a new class may share behaviour of a parent class, but may need to modify it
- Want to adapt features from **PERSON** that may not be quite appropriate for its subclasses



Eiffel Adaptation Mechanisms

- Renaming
  - » **Rename P as Q**
    - > **Change the name of a feature from P to Q**
- Redefining
  - » **feature behaviour**
- Changing
  - » **export permissions**
- Effecting
  - » **implementing deferred features**
- Undefine
  - » **When a feature is not needed -- makes class deferred**

Redefinition

- Consider class PERSON with a feature display

Redefinition – 2

- Consider class PERSON with a feature display
- Display mechanisms may not be appropriate for subclasses – different objects to display depending upon type
  > **Want to change semantics not syntax**

Redefinition – 3

- Consider class PERSON with a feature display
- Display mechanisms may not be appropriate for subclasses – different objects to display depending upon type
    - > **Want to change semantics not syntax**

```
class EMPLOYEE inherit PERSON
redefine display end
...
display do
   -- new display body here
end
...
end
```

Constraints on Redefinition

- You do not have complete freedom with redefinition

Constraints on Redefinition – 2

- You do not have complete freedom with redefinition

- Rules have to be obeyed in order to maintain **substitutability** and **strong typing**

Constraints on Redefinition – 3

- You do not have complete freedom with redefinition

- Rules have to be obeyed in order to maintain **substitutability** and **strong typing**

- If you change a type in a redefinition it must be a subtype of the original

Constraints on Redefinition – 4

- You do not have complete freedom with redefinition

- Rules have to be obeyed in order to maintain **substitutability** and **strong typing**

- If you change a type in a redefinition it must be a subtype of the original
  - » **Within that constraint, can change**
    - > **result type**
    - > **parameter types**

Eiffel Redefinition Rules

- Function with no arguments can be redefined to an attribute but **NOT** vice-versa

Eiffel Redefinition Rules – 2

- Function with no arguments can be redefined to an attribute but **NOT** vice-versa
  - » **Assignment possible for attributes, not functions**

Eiffel Redefinition Rules – 3

- Function with no arguments can be redefined to an attribute but **NOT** vice-versa
  - » **Assignment possible for attributes, not functions**

- Redefined feature must type conform to the original

Eiffel Redefinition Rules – 4

- Function with no arguments can be redefined to an attribute but **NOT** vice-versa
  - **Assignment possible for attributes, not functions**

- Redefined feature must type conform to the original

- Redefined feature must conform with respect to correctness to the original
  - **See this when we get to inheritance and contracts**

Eiffel Redefinition Rules – 5

- Prefix a feature with **frozen** to prevent redefinition

Eiffel Redefinition Rules – 6

- Prefix a feature with **frozen** to prevent redefinition

- To execute the original definition within the redefinition use

  Precursor { parent_class } (...)

Eiffel Redefinition Rules – 7

- Prefix a feature with **frozen** to prevent redefinition

- To execute the original definition within the redefinition use

  Precursor { parent_class } (...)

  » **Similar to super in Java**

Eiffel Redefinition Rules – 8

- Prefix a feature with **frozen** to prevent redefinition

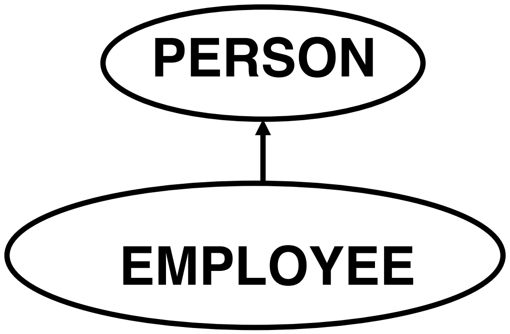
- To execute the original definition within the redefinition use

  Precursor { parent_class } (...)

  » **Similar to super in Java**

  » **Parent_class is used only for multiple inheritance to disambiguate which parent**

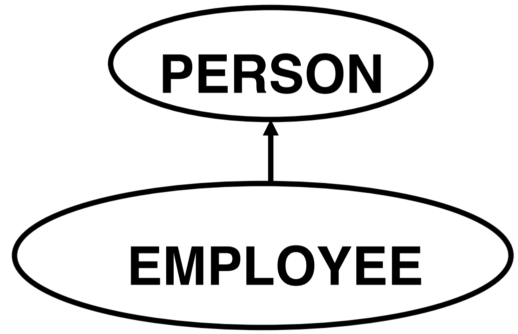
Renaming vs Redefinition




```
class EMPLOYEE
  inherit PERSON
  rename
    display as p_display

  feature { ANY }
     display do ... end

end
```




```
class EMPLOYEE
  inherit PERSON
  redefine
    display end

  feature { ANY }
     display do ... end

end
```

Notes – Renaming vs Redefinition

- Renaming
  - » **no formal connection between display features even though they have the same name**
  - » **Can change the contract !**
- Redefining
  - » **close connection between display features**
- Using redefinition
  - » **Essential for successful use of dynamic binding**
  - » **Cannot change the contract !**

Redefining a Signature

- May change a signature to maintain correctness
- Consider a DEVICE class used to represent hardware that can be attached to a network.
  - » **For every device there is an alternate – used when the first is not available**

```
class DEVICE feature
   alternate : DEVICE
   set_alternate ( a : DEVICE )
      do
         alternate := a
      end
   ....
end
```

Redefining a Signature – 2

- A PRINTER is a special kind of DEVICE
  > should inherit from DEVICE but alternate can only be another PRINTER

```
class PRINTER inherit DEVICE
  redefine alternate, set_alternate end
  feature
    alternate : PRINTER
    set_alternate ( a : PRINTER )
        do
            alternate := a
        end
    ....
end
```

Types have changed from DEVICE to PRINTER

PRINTER is a subtype of DEVICE

All is well

Type Redeclaration Rule

A redeclaration of a feature may replace the type of the feature (in an attribute or function) or the type of a formal argument (if a routine) by any type that conforms to the original

» **See *Redefining a Signature* slides**

Type Redeclaration Problem

- While the rule guarantees proper typing inconsistencies can arise if types are not changed consistently

  » **Leads to use of Anchored Declarations**

    > **The ability to define types relatively and not absolutely**

Anchored Declaration

- Provide a shortcut for certain kinds of signature redefinitions
- Declarations can be made relative to an **anchor type** rather than providing an absolute declaration

```
class NODE [ G ]   create make

feature { NONE }
      item : G  -- what's held in the node
      next : like Current
feature { ANY }
      make (g : G ) ...
      change_item ( g : G )
      change_next ( other : like next )

end
```

Current is the anchor. next points to a node of the same type as Current

other is same type as Next – recursive to Current

Anchored Declaration Rules

- The base class of **like anchor** is
  - **the base class of the type of anchor in the current class**
  - **If anchor is Current, then the base class is the enclosing class**

Anchored Declaration Rules – 2

- The base class of **like anchor** is
  - » **the base class of the type of anchor in the current class**
  - » **If anchor is Current, then the base class is the enclosing class**
- Can have recursive definition
  - » **like anchor can be based on an anchored type**
  - » **Do not have cycles in the anchor chain – no knots**

Anchored Declaration Rules – 3

- The base class of **like anchor** is
  - » **the base class of the type of anchor in the current class**
  - » **If anchor is Current, then the base class is the enclosing class**
- Can have recursive definition
  - » **like anchor can be based on an anchored type**
  - » **Do not have cycles in the anchor chain – no knots**
- While **like anchor** conforms to its base class **T**, **T** does not conform to **like anchor**
  - » **Problems occur if the anchor is redeclared in a subclass (see warning p603 CD, p604 book)**

Information Hiding and Inheritance

- Inheritance and Information Hiding are orthogonal mechanisms

  » **If B inherits from A**

    > **B is free to export or hide any feature it inherits in all possible combinations**

Information Hiding and Inheritance – 2

- Inheritance and Information Hiding are orthogonal mechanisms
  - » **If B inherits from A**
    - > **B is free to export or hide any feature it inherits in all possible combinations**
  - » **Need an export clause to change the export status from that of the parent**

```
class B inherit
    A
        export { NONE } f end      -- f is secret
        export { ANY } g end       -- g is public
        export { X, Y } h end      -- h is selectively public
...                                -- to X, Y  and their descendants
end
```

Interface & Implementation Use

Client	Inheritance
Use through interface	Use of implementation
Information hiding	No information hiding
Protection against changes in original implementation	No protection against changes in original implementation

Deferred Features and Classes

- Do not need nor always can define everything (fully implement) within a class

Deferred Features and Classes – 2

- Do not need nor always can define everything (fully implement) within a class
- Consider the **FIGURE** hierarchy
  - » **Most general notion is FIGURE**

Deferred Features and Classes – 3

- Do not need nor always can define everything (fully implement) within a class
- Consider the **FIGURE** hierarchy
  - » **Most general notion is FIGURE**
- Ideally want to apply **rotate** and **translate** to any figure **f** letting dynamic binding select the appropriate method at run time

Deferred Features and Classes – 4

- Do not need nor always can define everything (fully implement) within a class
- Consider the **FIGURE** hierarchy
  - » **Most general notion is FIGURE**
- Ideally want to apply **rotate** and **translate** to any figure **f** letting dynamic binding select the appropriate method at run time
- Could define a rotate, but useless
  - » **There is nothing to define**
  - » **Figure cannot provide even a default implementation**

Deferred Features and Classes – 5

- Want to declare the existence of **rotate** and translate at the **FIGURE** level so all subtypes have these features available

Deferred Features and Classes – 6

- Want to declare the existence of **rotate** and translate at the **FIGURE** level so all subtypes have these features available
- Let the actual descendants provide the specific implementation each type needs

Deferred Features and Classes – 7

- Want to declare the existence of **rotate** and translate at the **FIGURE** level so all subtypes have these features available
- Let the actual descendants provide the specific implementation each type needs
- Such features are called **deferred** and classes containing at least one deferred feature are called **deferred classes**

```
rotate ( centre : POINT ; angle : REAL )
    deferred
end
```

Effecting as feature

- In a proper descendent of **FIGURE** you will need to implement rotate
  - » **Process is called effecting**

Effecting as feature – 2

- In a proper descendent of **FIGURE** you will need to implement rotate
  - » **Process is called effecting**
- Deferred features are not redefined as there is no definition to modify
  - > **Instead we redeclare them**

```
class POLYGON inherit FIGURE
   feature
   rotate ( centre : POINT ; angle : REAL )
      -- write the rotation algorithm here
   end
...
end
```

Undefining a feature

- Used when a feature is defined in a parent class but not needed or wanted in a child class

  > **Useful in multiple inheritance**

Undefining a feature – 2

- Used when a feature is defined in a parent class but not needed or wanted in a child class

    > **Useful in multiple inheritance**

- Undefining properties
    - » **Feature is not usable in a child class**
    - » **We still have substitutability**
    - » **Cannot call an undefined feature**

Undefining a feature – 3

- What if we call an undefined feature?
  - » **Undefining makes an effective feature deferred**

```
deferred class CIRCLE inherit ELLIPSE
   undefine rotate end
...
end
```

Cannot instantiate a circle – has a deferred method

Redeclaration Table

Redeclaring from → / to ↓	Deferred	Effective
Deferred	Redefine	Undefine
Effective	Redeclare	Redefine

Types and Modules – Dual Perspective