

Inheritance

Readings: OOSCS2 Chapters 14 – 16



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG



Aspects of Inheritance

- **Code Reuse**
- Substitutability
 - **Polymorphism** and **Dynamic Binding** [compile-time type checks]
 - **Sub-contracting** [runtime behaviour checks]

3 of 60

Learning Objectives



Upon completing this lecture, you are expected to understand:

1. Design Attempts without Inheritance (w.r.t. Cohesion, SCP)
2. Using Inheritance for Code Reuse
3. Static Type & Polymorphism
4. Dynamic Type & Dynamic Binding
5. Type Casting
6. Polymorphism & Dynamic Binding:
Routine Arguments, Routine Return Values, Collections

2 of 60

Why Inheritance: A Motivating Example



Problem: A *student management system* stores data about students. There are two kinds of university students: *resident* students and *non-resident* students. Both kinds of students have a *name* and a list of *registered courses*. Both kinds of students are restricted to *register* for no more than 30 courses. When *calculating the tuition* for a student, a base amount is first determined from the list of courses they are currently registered (each course has an associated fee). For a non-resident student, there is a *discount rate* applied to the base amount to waive the fee for on-campus accommodation. For a resident student, there is a *premium rate* applied to the base amount to account for the fee for on-campus accommodation and meals.

Tasks: Design classes that satisfy the above problem statement. At runtime, each type of student must be able to register a course and calculate their tuition fee.

4 of 60

The COURSE Class



```
class
  COURSE

create -- Declare commands that can be used as constructors
  make

feature -- Attributes
  title: STRING
  fee: REAL

feature -- Commands
  make (t: STRING; f: REAL)
    -- Initialize a course with title 't' and fee 'f'.
    do
      title := t
      fee := f
    end
end
```

5 of 60

No Inheritance: NON_RESIDENT_STUDENT Class



```
class NON_RESIDENT_STUDENT
  create make
  feature -- Attributes
    name: STRING
    courses: LINKED_LIST[COURSE]
    discount_rate: REAL
  feature -- Constructor
    make (n: STRING)
      do name := n ; create courses.make end
  feature -- Commands
    set_dr (r: REAL) do discount_rate := r end
    register (c: COURSE) do courses.extend (c) end
  feature -- Queries
    tuition: REAL
    local base: REAL
    do base := 0.0
      across courses as c loop base := base + c.item.fee end
    Result := base * discount_rate
  end
end
```

7 of 60

No Inheritance: RESIDENT_STUDENT Class



```
class RESIDENT_STUDENT
  create make
  feature -- Attributes
    name: STRING
    courses: LINKED_LIST[COURSE]
    premium_rate: REAL
  feature -- Constructor
    make (n: STRING)
      do name := n ; create courses.make end
  feature -- Commands
    set_pr (r: REAL) do premium_rate := r end
    register (c: COURSE) do courses.extend (c) end
  feature -- Queries
    tuition: REAL
    local base: REAL
    do base := 0.0
      across courses as c loop base := base + c.item.fee end
    Result := base * premium_rate
  end
end
```

6 of 60

No Inheritance: Testing Student Classes



```
test_students: BOOLEAN
  local
    c1, c2: COURSE
    jim: RESIDENT_STUDENT
    jeremy: NON_RESIDENT_STUDENT
  do
    create c1.make ("EECS2030", 500.0)
    create c2.make ("EECS3311", 500.0)
    create jim.make ("J. Davis")
    jim.set_pr (1.25)
    jim.register (c1)
    jim.register (c2)
    Result := jim.tuition = 1250
    check Result end
    create jeremy.make ("J. Gibbons")
    jeremy.set_dr (0.75)
    jeremy.register (c1)
    jeremy.register (c2)
    Result := jeremy.tuition = 750
  end
```

8 of 60

No Inheritance: Issues with the Student Classes



- Implementations for the two student classes seem to work. But can you see any potential problems with it?
- The code of the two student classes share a lot in common.
- **Duplicates of code make it hard to maintain your software!**
- This means that when there is a change of policy on the common part, we need modify *more than one places*.
⇒ This violates the **Single Choice Principle**:
when a **change** is needed, there should be **a single place** (or **a minimal number of places**) where you need to make that change.

9 of 60

No Inheritance: Maintainability of Code (2)



What if a **new** way for base tuition calculation is to be implemented?

e.g.,

```
tuition: REAL
  local base: REAL
  do base := 0.0
    across courses as c loop base := base + c.item.fee end
  Result := base * inflation_rate * ...
end
```

We need to change the `tuition` query in **both** student classes.

⇒ **Violation** of the **Single Choice Principle**

11 of 60

No Inheritance: Maintainability of Code (1)



What if a **new** way for course registration is to be implemented?

e.g.,

```
register(Course c)
do
  if courses.count >= MAX_CAPACITY then
    -- Error: maximum capacity reached.
  else
    courses.extend (c)
  end
end
```

We need to change the `register` commands in **both** student classes!

⇒ **Violation** of the **Single Choice Principle**

10 of 60

No Inheritance: A Collection of Various Kinds of Students



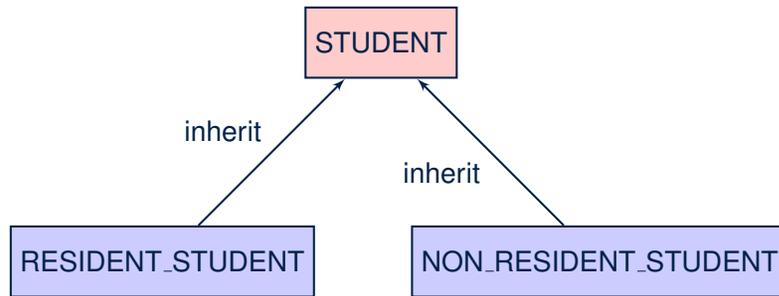
How do you define a class `StudentManagementSystem` that contains a list of **resident** and **non-resident** students?

```
class STUDENT_MANAGEMENT_SYSETM
  rs : LINKED_LIST[RESIDENT_STUDENT]
  nrs : LINKED_LIST[NON_RESIDENT_STUDENT]
  add_rs (rs: RESIDENT_STUDENT) do ... end
  add_nrs (nrs: NON_RESIDENT_STUDENT) do ... end
  register_all (Course c) -- Register a common course 'c'.
  do
    across rs as c loop c.item.register (c) end
    across nrs as c loop c.item.register (c) end
  end
end
```

But what if we later on introduce **more kinds of students**?
Inconvenient to handle each list of students, in pretty much the **same** manner, **separately**!

12 of 60

Inheritance Architecture



13 of 60

Inheritance: The STUDENT Parent Class



```
1 class STUDENT
2 create make
3 feature -- Attributes
4   name: STRING
5   courses: LINKED_LIST[COURSE]
6 feature -- Commands that can be used as constructors.
7   make (n: STRING) do name := n ; create courses.make end
8 feature -- Commands
9   register (c: COURSE) do courses.extend (c) end
10 feature -- Queries
11   tuition: REAL
12   local base: REAL
13   do base := 0.0
14     across courses as c loop base := base + c.item.fee end
15   Result := base
16 end
17 end
```

14 of 60

Inheritance: The RESIDENT_STUDENT Child Class



```
1 class
2   RESIDENT_STUDENT
3 inherit
4   STUDENT
5   redefine tuition end
6 create make
7 feature -- Attributes
8   premium_rate: REAL
9 feature -- Commands
10  set_pr (r: REAL) do premium_rate := r end
11 feature -- Queries
12   tuition: REAL
13   local base: REAL
14   do base := Precursor ; Result := base * premium_rate end
15 end
```

- L3: RESIDENT_STUDENT inherits all features from STUDENT.
- There is no need to repeat the register command
- L14: *Precursor* returns the value from query tuition in STUDENT.

15 of 60

Inheritance: The NON_RESIDENT_STUDENT Child Class

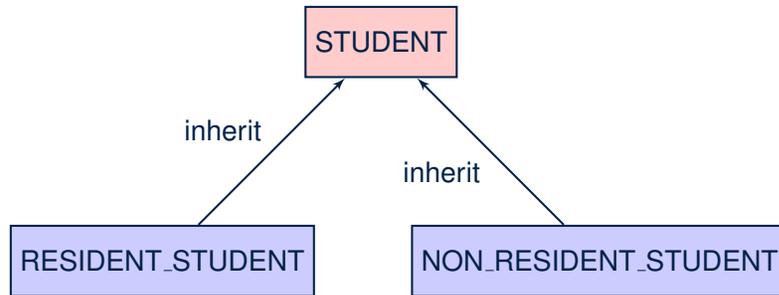


```
1 class
2   NON_RESIDENT_STUDENT
3 inherit
4   STUDENT
5   redefine tuition end
6 create make
7 feature -- Attributes
8   discount_rate: REAL
9 feature -- Commands
10  set_dr (r: REAL) do discount_rate := r end
11 feature -- Queries
12   tuition: REAL
13   local base: REAL
14   do base := Precursor ; Result := base * discount_rate end
15 end
```

- L3: NON_RESIDENT_STUDENT inherits all features from STUDENT.
- There is no need to repeat the register command
- L14: *Precursor* returns the value from query tuition in STUDENT.

16 of 60

Inheritance Architecture Revisited



- The class that defines the common features (attributes, commands, queries) is called the **parent**, **super**, or **ancestor** class.
- Each “specialized” class is called a **child**, **sub**, or **descendent** class.

17 of 60

Testing the Two Student Sub-Classes



```
test_students: BOOLEAN
local
  c1, c2: COURSE
  jim: RESIDENT_STUDENT ; jeremy: NON_RESIDENT_STUDENT
do
  create c1.make ("EECS2030", 500.0); create c2.make ("EECS3311", 500.0)
  create jim.make ("J. Davis")
  jim.set_pr (1.25) ; jim.register (c1); jim.register (c2)
  Result := jim.tuition = 1250
check Result end
  create jeremy.make ("J. Gibbons")
  jeremy.set_dr (0.75); jeremy.register (c1); jeremy.register (c2)
  Result := jeremy.tuition = 750
end
```

- The software can be used in exactly the same way as before (because we did not modify **feature signatures**).
- But now the internal structure of code has been made **maintainable** using **inheritance**.

19 of 60

Using Inheritance for Code Reuse



Inheritance in Eiffel (or any OOP language) allows you to:

- Factor out **common features** (attributes, commands, queries) in a separate class.
e.g., the `STUDENT` class
- Define an “specialized” version of the class which:
 - **inherits** definitions of all attributes, commands, and queries
e.g., attributes `name`, `courses`
e.g., command `register`
e.g., query on base amount in `tuition`
This means code reuse and elimination of code duplicates!
 - **defines new** features if necessary
e.g., `set_pr` for `RESIDENT_STUDENT`
e.g., `set_dr` for `NON_RESIDENT_STUDENT`
 - **redefines** features if necessary
e.g., compounded tuition for `RESIDENT_STUDENT`
e.g., discounted tuition for `NON_RESIDENT_STUDENT`

18 of 60

Static Type vs. Dynamic Type



- In **object orientation**, an entity has two kinds of types:
 - **static type** is declared at compile time [**unchangeable**]
An entity's **ST** determines what features may be called upon it.
 - **dynamic type** is changeable at runtime

- In Java:

```
Student s = new Student("Alan");
Student rs = new ResidentStudent("Mark");
```

- In Eiffel:

```
local s: STUDENT
      rs: STUDENT
do create {STUDENT} s.make ("Alan")
   create {RESIDENT_STUDENT} rs.make ("Mark")
```

- In Eiffel, the **dynamic type** can be omitted if it is meant to be the same as the **static type**:

```
local s: STUDENT
do create s.make ("Alan")
```

20 of 60

Inheritance Architecture Revisited



```

register (c: COURSE)+
tuition: REAL+
class STUDENT
name: STRING
courses: LINKED_LIST[COURSE]

/* new features */
premium_rate: REAL
set_pr (r: REAL)+
/* redefined features */
tuition: REAL++
class RESIDENT_STUDENT

/* new features */
discount_rate: REAL
set_dr (r: REAL)+
/* redefined features */
tuition: REAL++
class NON_RESIDENT_STUDENT
    
```

```

s1,s2,s3: STUDENT ; rs: RESIDENT_STUDENT ; nrs : NON_RESIDENT_STUDENT
create {STUDENT} s1.make ("S1")
create {RESIDENT_STUDENT} s2.make ("S2")
create {NON_RESIDENT_STUDENT} s3.make ("S3")
create {RESIDENT_STUDENT} rs.make ("RS")
create {NON_RESIDENT_STUDENT} nrs.make ("NRS")
    
```

	name	courses	reg	tuition	pr	set_pr	dr	set_dr
s1.		✓					✗	
s2.		✓					✗	
s3.		✓					✗	
rs.		✓			✓			✗
nrs.		✓			✗			✓

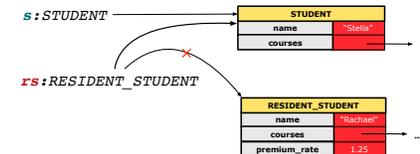
Polymorphism: Intuition (2)



```

1 local s: STUDENT ; rs: RESIDENT_STUDENT
2 do create {STUDENT} s.make ("Stella")
3   create {RESIDENT_STUDENT} rs.make ("Rachael")
4   rs.set_pr (1.25)
5   s := rs /* Is this valid? */
6   rs := s /* Is this valid? */
    
```

- rs := s (L6) should be *invalid*:



- rs declared of type RESIDENT_STUDENT ∴ calling rs.set_pr(1.50) can be expected.
- rs is now pointing to a STUDENT object.
- Then, what would happen to rs.set_pr(1.50)?
CRASH ∴ rs.premium_rate is *undefined*!!

Polymorphism: Intuition (1)



```

1 local
2   s: STUDENT
3   rs: RESIDENT_STUDENT
4 do
5   create s.make ("Stella")
6   create rs.make ("Rachael")
7   rs.set_pr (1.25)
8   s := rs /* Is this valid? */
9   rs := s /* Is this valid? */
    
```

- Which one of L8 and L9 is *valid*? Which one is *invalid*?
 - L8: What kind of address can s store? [STUDENT]
 - ∴ The context object s is expected to be used as:
 - s.register(eecs3311) and s.tuition
 - L9: What kind of address can rs store? [RESIDENT_STUDENT]
 - ∴ The context object rs is expected to be used as:
 - rs.register(eecs3311) and rs.tuition
 - rs.set_pr (1.50) [increase premium rate]

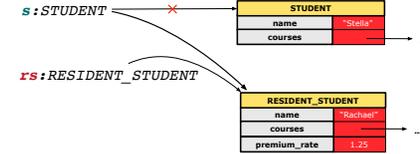
Polymorphism: Intuition (3)



```

1 local s: STUDENT ; rs: RESIDENT_STUDENT
2 do create {STUDENT} s.make ("Stella")
3   create {RESIDENT_STUDENT} rs.make ("Rachael")
4   rs.set_pr (1.25)
5   s := rs /* Is this valid? */
6   rs := s /* Is this valid? */
    
```

- s := rs (L5) should be *valid*:



- Since s is declared of type STUDENT, a subsequent call s.set_pr(1.50) is *never* expected.
- s is now pointing to a RESIDENT_STUDENT object.
- Then, what would happen to s.tuition?
OK ∴ s.premium_rate is just *never used*!!

Dynamic Binding: Intuition (1)

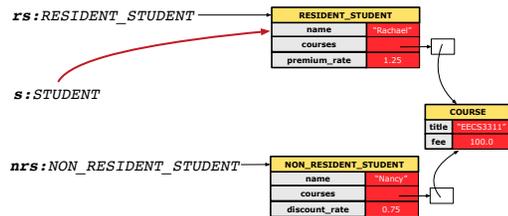


```

1 local c : COURSE ; s : STUDENT
2   rs : RESIDENT_STUDENT ; nrs : NON_RESIDENT_STUDENT
3 do create c.make ("EECS3311", 100.0)
4   create {RESIDENT_STUDENT} rs.make("Rachael")
5   create {NON_RESIDENT_STUDENT} nrs.make("Nancy")
6   rs.set_pr(1.25); rs.register(c)
7   nrs.set_dr(0.75); nrs.register(c)
8   s := rs; ; check s.tuition = 125.0 end
9   s := nrs; ; check s.tuition = 75.0 end

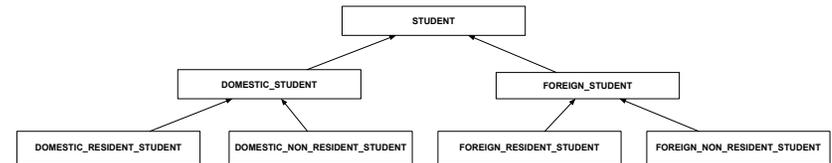
```

After `s := rs (L7)`, `s` points to a `RESIDENT_STUDENT` object.
 ⇒ Calling `s.tuition` applies the `premium_rate`.



25 of 60

Multi-Level Inheritance Architecture (1)



27 of 60

Dynamic Binding: Intuition (2)

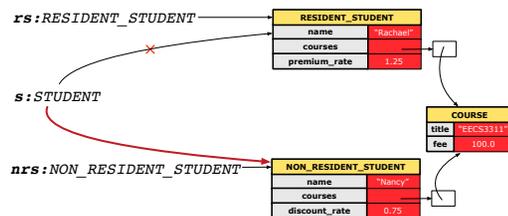


```

1 local c : COURSE ; s : STUDENT
2   rs : RESIDENT_STUDENT ; nrs : NON_RESIDENT_STUDENT
3 do create c.make ("EECS3311", 100.0)
4   create {RESIDENT_STUDENT} rs.make("Rachael")
5   create {NON_RESIDENT_STUDENT} nrs.make("Nancy")
6   rs.set_pr(1.25); rs.register(c)
7   nrs.set_dr(0.75); nrs.register(c)
8   s := rs; ; check s.tuition = 125.0 end
9   s := nrs; ; check s.tuition = 75.0 end

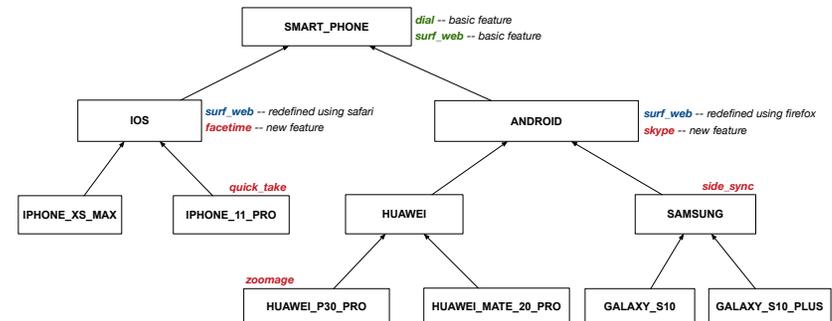
```

After `s := nrs (L8)`, `s` points to a `NON_RESIDENT_STUDENT` object.
 ⇒ Calling `s.tuition` applies the `discount_rate`.



26 of 60

Multi-Level Inheritance Architecture (2)



28 of 60

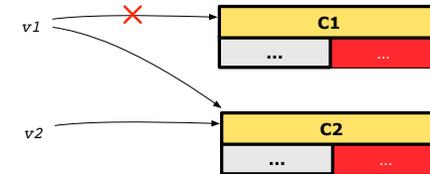
Inheritance Forms a Type Hierarchy

- A (data) **type** denotes a set of related *runtime values*.
 - Every *class* can be used as a type: the set of runtime *objects*.
- Use of *inheritance* creates a **hierarchy** of classes:
 - (Implicit) Root of the hierarchy is ANY.
 - Each *inherit* declaration corresponds to an upward arrow.
 - The *inherit* relationship is *transitive*: when A inherits B and B inherits C, we say A *indirectly* inherits C.
 - e.g., Every class implicitly *inherits* the ANY class.
- Ancestor** vs. **Descendant** classes:
 - The **ancestor classes** of a class A are: A itself and all classes that A directly, or indirectly, inherits.
 - A *inherits* all features from its *ancestor classes*.
 - ∴ A's instances have a **wider range of expected usages** (i.e., attributes, queries, commands) than instances of its *ancestor* classes.
 - The **descendant classes** of a class A are: A itself and all classes that directly, or indirectly, inherits A.
 - Code defined in A is *inherited* to all its *descendant classes*.

29 of 60

Substitutions via Assignments

- By declaring $v1 : C1$, *reference variable* v1 will store the *address* of an object of class C1 at runtime.
- By declaring $v2 : C2$, *reference variable* v2 will store the *address* of an object of class C2 at runtime.
- Assignment $v1 := v2$ *copies the address* stored in v2 into v1.
 - v1 will instead point to wherever v2 is pointing to. [**object alias**]



- In such assignment $v1 := v2$, we say that we **substitute** an object of type C1 with an object of type C2.
- Substitutions** are subject to *rules!*

31 of 60

Inheritance Accumulates Code for Reuse

- The *lower* a class is in the type hierarchy, the *more code* it accumulates from its *ancestor classes*:
 - A *descendant class* inherits all code from its *ancestor classes*.
 - A *descendant class* may also:
 - Declare new attributes.
 - Define new queries or commands.
 - Redefine** inherited queries or commands.
- Consequently:
 - When being used as **context objects**, instances of a class' *descendant classes* have a **wider range of expected usages** (i.e., attributes, commands, queries).
 - When expecting an object of a particular class, we may **substitute** it with an object of any of its *descendant classes*.
 - e.g., When expecting a STUDENT object, substitute it with either a RESIDENT_STUDENT or a NON_RESIDENT_STUDENT object.
 - Justification:** A *descendant class* contains **at least as many** features as defined in its *ancestor classes* (but **not vice versa!**).

30 of 60

Rules of Substitution

Given an inheritance hierarchy:

- When expecting an object of class A, it is *safe* to **substitute** it with an object of any **descendant class** of A (including A).
 - e.g., When expecting an IOS phone, you *can* substitute it with either an IPHONE_XS_MAX or IPHONE_11_PRO.
 - ∴ Each **descendant class** of A is guaranteed to contain all code of (non-private) attributes, commands, and queries defined in A.
 - ∴ All features defined in A are *guaranteed to be available* in the new substitute.
- When expecting an object of class A, it is *unsafe* to **substitute** it with an object of any **ancestor class of A's parent**.
 - e.g., When expecting an IOS phone, you *cannot* substitute it with just a SMART_PHONE, because the facetime feature is not supported in an ANDROID phone.
 - ∴ Class A may have defined new features that do not exist in any of its **parent's ancestor classes**.

32 of 60

Reference Variable: Static Type

- A reference variable's **static type** is what we declare it to be.
 - e.g., `jim:STUDENT` declares jim's static type as STUDENT.
 - e.g., `my_phone:SMART_PHONE` declares a variable `my_phone` of static type `SmartPhone`.
 - The **static type** of a reference variable **never changes**.
- For a reference variable `v`, its **static type** `C` defines the **expected usages of `v` as a context object**.
- A feature call `v.m(...)` is **compilable** if `m` is defined in `C`.
 - e.g., After declaring `jim:STUDENT`, we
 - **may** call `register` and `tuition` on `jim`
 - **may not** call `set_pr` (specific to a resident student) or `set_dr` (specific to a non-resident student) on `jim`
 - e.g., After declaring `my_phone:SMART_PHONE`, we
 - **may** call `dial` and `surf_web` on `my_phone`
 - **may not** call `facetime` (specific to an IOS phone) or `skype` (specific to an Android phone) on `my_phone`

33 of 60

Reference Variable: Changing Dynamic Type (1)

Re-assigning a reference variable to a newly-created object:

- **Substitution Principle**: the new object's class must be a **descendant class** of the reference variable's **static type**.
- e.g., Given the declaration `jim:STUDENT`:
 - `create {RESIDENT_STUDENT} jim.make("Jim")` changes the **dynamic type** of `jim` to `RESIDENT_STUDENT`.
 - `create {NON_RESIDENT_STUDENT} jim.make("Jim")` changes the **dynamic type** of `jim` to `NON_RESIDENT_STUDENT`.
- e.g., Given an alternative declaration `jim:RESIDENT_STUDENT`:
 - e.g., `create {STUDENT} jim.make("Jim")` is illegal because `STUDENT` is **not** a **descendant class** of the **static type** of `jim` (i.e., `RESIDENT_STUDENT`).

35 of 60

Reference Variable: Dynamic Type

A reference variable's **dynamic type** is the type of object that it is currently pointing to at runtime.

- The **dynamic type** of a reference variable **may change** whenever we **re-assign** that variable to a different object.
- There are two ways to re-assigning a reference variable.

34 of 60

Reference Variable: Changing Dynamic Type (2)

Re-assigning a reference variable `v` to an existing object that is referenced by another variable `other` (i.e., `v := other`):

- **Substitution Principle**: the static type of `other` must be a **descendant class** of `v`'s **static type**.
- e.g.,

```
jim: STUDENT ; rs: RESIDENT_STUDENT; nrs: NON_RESIDENT_STUDENT
create {STUDENT} jim.make (...)
create {RESIDENT_STUDENT} rs.make (...)
create {NON_RESIDENT_STUDENT} nrs.make (...)
```

- `rs := jim` ✗
- `nrs := jim` ✗
- `jim := rs` ✓
changes the **dynamic type** of `jim` to the dynamic type of `rs`
- `jim := nrs` ✓
changes the **dynamic type** of `jim` to the dynamic type of `nrs`

36 of 60

Polymorphism and Dynamic Binding (1)



- Polymorphism**: An object variable may have “multiple possible shapes” (i.e., allowable **dynamic types**).
 - Consequently, there are **multiple possible versions** of each feature that may be called.
 - e.g., 3 possibilities of tuition on a **STUDENT** reference variable:
 - In **STUDENT**: base amount
 - In **RESIDENT_STUDENT**: base amount with `premium_rate`
 - In **NON_RESIDENT_STUDENT**: base amount with `discount_rate`
- Dynamic binding**: When a feature `m` is called on an object variable, the version of `m` corresponding to its “current shape” (i.e., one defined in the **dynamic type** of `m`) will be called.

```

jim: STUDENT; rs: RESIDENT_STUDENT; nrs: NON_STUDENT
create {RESIDENT_STUDENT} rs.make (...)
create {NON_RESIDENT_STUDENT} nrs.nrs (...)
jim := rs
jim.tuition; /* version in RESIDENT_STUDENT */
jim := nrs
jim.tuition; /* version in NON_RESIDENT_STUDENT */
    
```

37 of 60

Polymorphism and Dynamic Binding (2.2)



```

test_dynamic_binding_students: BOOLEAN
local
  jim: STUDENT
  rs: RESIDENT_STUDENT
  nrs: NON_RESIDENT_STUDENT
  c: COURSE
do
  create c.make ("ECS3311", 500.0)
  create {STUDENT} jim.make ("J. Davis")
  create {RESIDENT_STUDENT} rs.make ("J. Davis")
  rs.register (c)
  rs.set_pr (1.5)
  jim := rs
  Result := jim.tuition = 750.0
check Result end
  create {NON_RESIDENT_STUDENT} nrs.make ("J. Davis")
  nrs.register (c)
  nrs.set_dr (0.5)
  jim := nrs
  Result := jim.tuition = 250.0
end
    
```

39 of 60

Polymorphism and Dynamic Binding (2.1)



```

1 test_polymorphism_students
2 local
3   jim: STUDENT
4   rs: RESIDENT_STUDENT
5   nrs: NON_RESIDENT_STUDENT
6 do
7   create {STUDENT} jim.make ("J. Davis")
8   create {RESIDENT_STUDENT} rs.make ("J. Davis")
9   create {NON_RESIDENT_STUDENT} nrs.make ("J. Davis")
10  jim := rs ✓
11  rs := jim ✗
12  jim := nrs ✓
13  rs := jim ✗
14 end
    
```

In (L3, L7), (L4, L8), (L5, L9), **ST** = **DT**, so we may abbreviate:

L7: `create jim.make ("J. Davis")`

L8: `create rs.make ("J. Davis")`

L9: `create nrs.make ("J. Davis")`

38 of 60

Reference Type Casting: Motivation



```

1 local jim: STUDENT; rs: RESIDENT_STUDENT
2 do create {RESIDENT_STUDENT} jim.make ("J. Davis")
3   rs := jim
4   rs.setPremiumRate(1.5)
    
```

- Line 2 is **legal**: `RESIDENT_STUDENT` is a **descendant class** of the static type of `jim` (i.e., `STUDENT`).
- Line 3 is **illegal**: `jim`'s static type (i.e., `STUDENT`) is **not** a **descendant class** of `rs`'s static type (i.e., `RESIDENT_STUDENT`).
- Eiffel compiler is **unable to infer** that `jim`'s **dynamic type** in Line 4 is `RESIDENT_STUDENT`. [**Undecidable**]
- Force the Eiffel compiler to believe so, by replacing L3, L4 by a **type cast** (which **temporarily** changes the **ST** of `jim`):

```

check attached {RESIDENT_STUDENT} jim as rs_jim then
  rs := rs_jim
  rs.set_pr (1.5)
end
    
```

40 of 60

Reference Type Casting: Syntax

```

1 check attached {RESIDENT_STUDENT} jim as rs_jim then
2   rs := rs_jim
3   rs.set_pr (1.5)
4 end

```

L1 is an assertion:

- `attached RESIDENT_STUDENT jim` is a Boolean expression that is to be evaluated at **runtime**.
 - If it evaluates to **true**, then the `as rs_jim` expression has the effect of assigning “the cast version” of jim to a new variable rs_jim.
 - If it evaluates to **false**, then a runtime assertion violation occurs.
- **Dynamic Binding**: **Line 4** executes the correct version of set_pr.
- It is approximately the same as following Java code:

```

if(jim instanceof ResidentStudent) {
    ResidentStudent rs = (ResidentStudent) jim;
    rs.set_pr(1.5);
}
else { throw new Exception("Cast Not Done."); }

```

41 of 60

Notes on Type Cast (2)

- Given **v** of static type **ST**, it is **violation-free** to cast **v** to **C**, as long as **C** is a descendant or ancestor class of **ST**.
- Why Cast?
 - Without cast, we can **only** call features defined in **ST** on **v**.
 - By casting **v** to **C**, we create an **alias** of the object pointed by **v**, with the new **static type C**.
 - ⇒ All features that are defined in **C** can be called.

```

my_phone: IOS
create {IPHONE_11_PRO} my_phone.make
-- can only call features defined in IOS on myPhone
-- dial, surf_web, facetime ✓ quick_take, skype, side_sync, zoomage ×
check attached {SMART_PHONE} my_phone as sp then
-- can now call features defined in SMART_PHONE on sp
-- dial, surf_web ✓ facetime, quick_take, skype, side_sync, zoomage ×
end
check attached {IPHONE_11_PRO} my_phone as ip11_pro then
-- can now call features defined in IPHONE_11_PRO on ip11_pro
-- dial, surf_web, facetime, quick_take ✓ skype, side_sync, zoomage ×
end

```

43 of 60

Notes on Type Cast (1)

- `check attached {C} y then ... end` **always compiles**
- What if **C** is not an **ancestor** of **y**'s **DT**?
 - ⇒ A **runtime** assertion violation occurs!
 - ∴ **y**'s **DT** cannot fulfill the expectation of **C**.

42 of 60

Notes on Type Cast (3)

A cast `check attached {C} v as ...` triggers an **assertion violation** if **C** is **not** along the **ancestor path** of **v**'s **DT**.

```

test_smart_phone_type_cast_violation
local mine: ANDROID
do create {HUAWEI} mine.make
-- ST of mine is ANDROID; DT of mine is HUAWEI
check attached {SMART_PHONE} mine as sp then ... end
-- ST of sp is SMART_PHONE; DT of sp is HUAWEI
check attached {HUAWEI} mine as huawei then ... end
-- ST of huawei is HUAWEI; DT of huawei is HUAWEI
check attached {SAMSUNG} mine as samsung then ... end
-- Assertion violation
-- ∴ SAMSUNG is not ancestor of mine's DT (HUAWEI)
check attached {HUAWEI_P30_PRO} mine as p30_pro then ... end
-- Assertion violation
-- ∴ HUAWEI_P30_PRO is not ancestor of mine's DT (HUAWEI)
end

```

44 of 60

Polymorphism: Routine Call Parameters



```
1 class STUDENT_MANAGEMENT_SYSTEM {
2   ss : ARRAY[STUDENT] -- ss[i] has static type Student
3   add_s (s: STUDENT) do ss[0] := s end
4   add_rs (rs: RESIDENT_STUDENT) do ss[0] := rs end
5   add_nrs (nrs: NON_RESIDENT_STUDENT) do ss[0] := nrs end
```

- **L4:** `ss[0] := rs` is valid. ∴ RHS's ST **RESIDENT_STUDENT** is a **descendant class** of LHS's ST **STUDENT**.
- Say we have a **STUDENT_MANAGEMENT_SYSETM** object `sms`:
 - ∴ **call by value**, `sms.add_rs(o)` attempts the following assignment (i.e., replace parameter `rs` by a copy of argument `o`):

```
rs := o
```

- Whether this argument passing is valid depends on `o`'s **static type**.

Rule: In the signature of a feature `m`, if the type of a parameter is class `C`, then we may call feature `m` by passing objects whose **static types** are `C`'s **descendants**.

45 of 60

Why Inheritance: A Polymorphic Collection of Students



How do you define a class **STUDENT_MANAGEMENT_SYSETM** that contains a list of **resident** and **non-resident** students?

```
class STUDENT_MANAGEMENT_SYSETM
  students: LINKED_LIST[STUDENT]
  add_student(s: STUDENT)
  do
    students.extend(s)
  end
  registerAll(c: COURSE)
  do
    across
      students as s
    loop
      s.item.register(c)
    end
  end
end
```

47 of 60

Polymorphism: Routine Call Arguments



```
test_polymorphism_feature_arguments
local
  s1, s2, s3: STUDENT
  rs: RESIDENT_STUDENT ; nrs: NON_RESIDENT_STUDENT
  sms: STUDENT_MANAGEMENT_SYSTEM
do
  create sms.make
  create {STUDENT} s1.make("s1")
  create {RESIDENT_STUDENT} s2.make("s2")
  create {NON_RESIDENT_STUDENT} s3.make("s3")
  create {RESIDENT_STUDENT} rs.make("rs")
  create {NON_RESIDENT_STUDENT} nrs.make("nrs")
  sms.add_s(s1) ✓ sms.add_s(s2) ✓ sms.add_s(s3) ✓
  sms.add_s(rs) ✓ sms.add_s(nrs) ✓
  sms.add_rs(s1) × sms.add_rs(s2) × sms.add_rs(s3) ×
  sms.add_rs(rs) ✓ sms.add_rs(nrs) ×
  sms.add_nrs(s1) × sms.add_nrs(s2) × sms.add_nrs(s3) ×
  sms.add_nrs(rs) × sms.add_nrs(nrs) ✓
end
```

46 of 60

Polymorphism and Dynamic Binding: A Polymorphic Collection of Students



```
test_sms_polymorphism: BOOLEAN
local
  rs: RESIDENT_STUDENT
  nrs: NON_RESIDENT_STUDENT
  c: COURSE
  sms: STUDENT_MANAGEMENT_SYSTEM
do
  create rs.make("Jim")
  rs.set_pr(1.5)
  create nrs.make("Jeremy")
  nrs.set_dr(0.5)
  create sms.make
  sms.add_s(rs)
  sms.add_s(nrs)
  create c.make("EECS3311", 500)
  sms.register_all(c)
  Result := sms.ss[1].tuition = 750 and sms.ss[2].tuition = 250
end
```

48 of 60

Polymorphism: Return Values (1)

```

1 class STUDENT_MANAGEMENT_SYSTEM {
2   ss: LINKED_LIST[STUDENT]
3   add_s (s: STUDENT)
4     do
5       ss.extend (s)
6     end
7   get_student(i: INTEGER): STUDENT
8     require 1 <= i and i <= ss.count
9     do
10      Result := ss[i]
11    end
12 end

```

- L2: **ST** of each stored item (`ss[i]`) in the list: [STUDENT]
- L3: **ST** of input parameter `s`: [STUDENT]
- L7: **ST** of return value (`Result`) of `get_student`: [STUDENT]
- L11: `ss[i]`'s **ST** is *descendant* of `Result`' **ST**.

Question: What can be the *dynamic type* of `s` after **Line 11**?

Answer: All descendant classes of `Student`.

49 of 60

Design Principle: Polymorphism

- When declaring an attribute `a: T`
 ⇒ Choose *static type* `T` which “accumulates” all features that you predict you will want to call on `a`.
 e.g., Choose `s: STUDENT` if you do not intend to be specific about which kind of student `s` might be.
 ⇒ Let *dynamic binding* determine at runtime which version of `tuition` will be called.
- What if after declaring `s: STUDENT` you find yourself often needing to **cast** `s` to `RESIDENT_STUDENT` in order to access `premium_rate`?

```
check attached {RESIDENT_STUDENT} s as rs then rs.set_pr(...) end
```

⇒ Your design decision should have been: `s: RESIDENT_STUDENT`

- Same design principle applies to:
 - Type of feature parameters:
 - Type of queries:

```
f(a: T)
```

```
q(...): T
```

51 of 60

Polymorphism: Return Values (2)

```

1 test_sms_polymorphism: BOOLEAN
2 local
3   rs: RESIDENT_STUDENT ; nrs: NON_RESIDENT_STUDENT
4   c: COURSE ; sms: STUDENT_MANAGEMENT_SYSTEM
5 do
6   create rs.make ("Jim") ; rs.set_pr (1.5)
7   create nrs.make ("Jeremy") ; nrs.set_dr (0.5)
8   create sms.make ; sms.add_s (rs) ; sms.add_s (nrs)
9   create c.make ("EECS3311", 500) ; sms.register_all (c)
10  Result :=
11    sms.get_student(1).tuition = 750
12    and sms.get_student(2).tuition = 250
13 end

```

- L11: `get_student(1)`'s dynamic type? [RESIDENT_STUDENT]
- L11: Version of `tuition`? [RESIDENT_STUDENT]
- L12: `get_student(2)`'s dynamic type? [NON_RESIDENT_STUDENT]
- L12: Version of `tuition`? [NON_RESIDENT_STUDENT]

50 of 60

Static Type vs. Dynamic Type: When to consider which?

- **Whether or not an OOP code compiles** depends only on the *static types* of relevant variables.
 ∴ Inferring the *dynamic type* statically is an *undecidable* problem that is inherently impossible to solve.
- **The behaviour of Eiffel code being executed at runtime**
 e.g., which version of the routine is called
 e.g., if a `check attached {...} as ... then ... end` assertion error will occur
 depends on the *dynamic types* of relevant variables.
 ⇒ Best practice is to visualize how objects are created (by drawing boxes) and variables are re-assigned (by drawing arrows).

52 of 60

Summary: Type Checking Rules



CODE	CONDITION TO BE TYPE CORRECT
<code>x := y</code>	y's ST a descendant of x's ST
<code>x.f(y)</code>	Feature f defined in x's ST y's ST a descendant of f's parameter's ST
<code>z := x.f(y)</code>	Feature f defined in x's ST y's ST a descendant of f's parameter's ST ST of m's return value a descendant of z's ST
<code>check attached {C} y</code>	Always compiles
<code>check attached {C} y as temp then x := temp end</code>	C a descendant of x's ST
<code>check attached {C} y as temp then x.f(temp) end</code>	Feature f defined in x's ST C a descendant of f's parameter's ST

Even if `check attached {C} y then ... end` always compiles,
a runtime assertion error occurs if C is not an **ancestor** of y's **DT**!

53 of 60

Beyond this lecture ...



- **Written Notes:** Static Types, Dynamic Types, Type Casts

https://www.eecs.yorku.ca/~jackie/teaching/lectures/2020/F/EECS3311/notes/EECS3311_F20_Notes_Static_Types_Cast.pdf

- **Recommended Exercise 1:**

Expand the student inheritance design ([here](#)) to reproduce the various fragments of polymorphism and dynamic binding.

- **Recommended Exercise 2:**

Create a new project (using [eiffel-new](#)) to reproduce the various fragments related to the running example of smart phones.

54 of 60

Index (1)



Learning Objectives

Aspects of Inheritance

Why Inheritance: A Motivating Example

The COURSE Class

No Inheritance: RESIDENT STUDENT Class

No Inheritance: NON RESIDENT STUDENT Class

No Inheritance: Testing Student Classes

No Inheritance:

Issues with the Student Classes

No Inheritance: Maintainability of Code (1)

No Inheritance: Maintainability of Code (2)

55 of 60

Index (2)



No Inheritance:

A Collection of Various Kinds of Students

Inheritance Architecture

Inheritance: The STUDENT Parent Class

Inheritance:

The RESIDENT STUDENT Child Class

Inheritance:

The NON RESIDENT STUDENT Child Class

Inheritance Architecture Revisited

Using Inheritance for Code Reuse

Testing the Two Student Sub-Classes

Static Type vs. Dynamic Type

56 of 60

Index (3)



Inheritance Architecture Revisited

Polymorphism: Intuition (1)

Polymorphism: Intuition (2)

Polymorphism: Intuition (3)

Dynamic Binding: Intuition (1)

Dynamic Binding: Intuition (2)

Multi-Level Inheritance Architecture (1)

Multi-Level Inheritance Architecture (2)

Inheritance Forms a Type Hierarchy

Inheritance Accumulates Code for Reuse

Substitutions via Assignments

57 of 60

Index (4)



Rules of Substitution

Reference Variable: Static Type

Reference Variable: Dynamic Type

Reference Variable:

Changing Dynamic Type (1)

Reference Variable:

Changing Dynamic Type (2)

Polymorphism and Dynamic Binding (1)

Polymorphism and Dynamic Binding (2.1)

Polymorphism and Dynamic Binding (2.2)

Reference Type Casting: Motivation

Reference Type Casting: Syntax

58 of 60

Index (5)



Notes on Type Cast (1)

Notes on Type Cast (2)

Notes on Type Cast (3)

Polymorphism: Routine Call Parameters

Polymorphism: Routine Call Arguments

Why Inheritance:

A Polymorphic Collection of Students

Polymorphism and Dynamic Binding:

A Polymorphic Collection of Students

Polymorphism: Return Values (1)

Polymorphism: Return Values (2)

Design Principle: Polymorphism

59 of 60

Index (6)



Static Type vs. Dynamic Type:

When to consider which?

Summary: Type Checking Rules

Beyond this lecture ...

60 of 60