EECS3311 Software Design Winter 2019
Static Types, Expectations, Dynamic Types, and Type Casts

Chen-Wei Wang

Contents

1 Inheritance Hierarchy

Consider the following definitions of Eiffel classes

```
class
  A
create
  make
feature
  make do end
feature
  a: INTEGER
end
```

```
class
  B
inherit
  A
create
  make
feature
  b: INTEGER
end
```

```
class
  C
inherit
  A
create
  make
feature
  c: INTEGER
end
```

```
class
  D
inherit
  C
create
  make
feature
  d: INTEGER
end
```

which form the class hierarchy as shown in Figure 1:

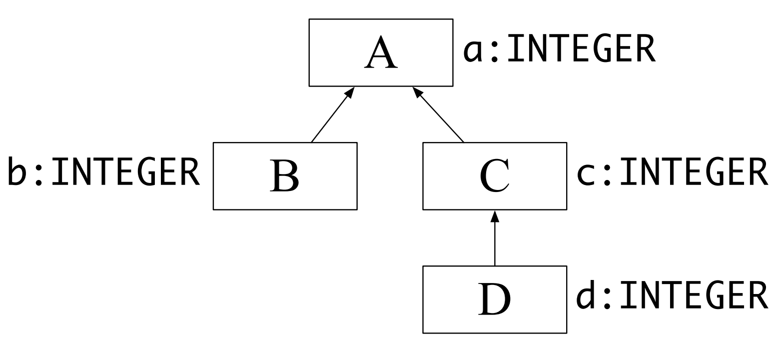

Figure 1: Class Inheritance Hierarchy

2 Static Types (at Compile Time) Define Expected Usages

Consider the following line of Eiffel code, which declares, at *compile time*, class `C` as the type of a reference variable `oc`:

```
oc: C
```

After the above declaration, we say that `C` is the ***static type*** of variable `oc`. The ***static type*** of variable `oc` constrains that, at *runtime*, `oc` stores the address of some `C` object. Consequently, only features (attributes, commands, and queries) that are defined and inherited in class `C` are expected to be called via `oc` as the context object:

- `oc.a`
- `oc.c`

Recall that a class only inherits code of features (i.e., attributes, commands, and queries) from its ancestor classes. Therefore, it is **not** expected to call:

- `oc.b` ($\because$ class `B` is not an ancestor class of `C`)
- `oc.d` ($\because$ class `D` is actually a child class of `C`)

From the inheritance hierarchy in Figure 1 (page 1), we have the following expectations for variables of the various types:

DECLARATION	EXPECTATIONS
`os: A`	`oa.a`
`ob: B`	`ob.a` `ob.b`
`oc: C`	`oc.a` `oc.c`

`od: D`	`od.a`
	`od.c`
	`od.d`

Figure 2: Declarations of Static Types and Expectations

3 Dynamic Types (at Runtime)

Because a reference variable's ***static type*** defines its expected usages at *runtime*, that variable's ***dynamic type*** must be consistent with the expectations. As an example, the following object attachments (i.e., object creations) are not valid:

```
1 oc1, oc2: C
2 create {A} oc1.make
3 create {B} oc2.make
```

Both of the above object attachments are **invalid**:

- For **Line 2**, if we allowed `oc1` to point to an `A` object (which only possesses the attribute `a`), then one of the expectations of `oc`, which is `oc.c` (see Figure 2), would not be met.

- Similarly, for **Line 3**, if we allowed `oc2` to point to a `B` object (which possesses attributes `a` and `b`), then one of the expectations of `oc`, which is `oc.c` (see Figure 2), would not be met.

Instead, the following object attachments are valid:

```
oc3, oc4: C
create {C} oc3.make
create {D} oc4.make
```

In the above object attachments, the expectations of ***static type*** `C` can be met by ***dynamic types*** `C` and `D`, which are both descendant classes of `C`.

4 Temporarily Changing the Static Type via a Cast

Always remember:

- To judge if a line of Eiffel code **compiles** or not, you **only** need to consider the ***static types*** of the variables involved (Section 4.1).
- To judge if a line of *compilable* Eiffel code causes an exception or violation at **runtime**, you need to then consider the ***dynamic types*** of the variable involved (Section 4.2).

4.1 Does a Cast Compile?

Principles:

- Casting a reference variable temporarily changes its ***static type***, and thus changes the expectations of that variable.
- A reference variable may be cast to any class that is either a descendant or an ancestor class of that variable's declared ***static type***.
- Casting a reference variable to a descendant class of its **widens** that variable's expectations (∵ a class' descendant class contains at least as many features).
- Symmetrically, casting a reference variable to a ancestor class of its **narrows** that variable's expectations.

For example, given a variable `oc` whose declared ***static type*** is `C` (i.e., `oc: C`), the following casts are compilable:

1. `check attached {D} oc as v then ... end` [`oc`'s scope is within ...]

   This cast creates a temporary variable `v` whose ***static type*** is `D`, and whose ***dynamic type*** is that of `oc`. Since `D` is a descendant class of `oc`'s ***static type*** (`C`), performing this cast **widens** the expectations: we can now expect `v.d`, whereas `oc.d` cannot be expected.
2. `check attached {C} oc as v then ... end` [`oc`'s scope is within ...]

   This cast creates a temporary variable `v` whose ***static type*** is `C`, and whose ***dynamic type*** is that of `oc`. Since `C` is both a descendant and an ancestor class of `oc`'s ***static type*** (`C`), performing this cast results in the same expectations: `v.a` and `v.c`.
3. `check attached {A} oc as v then ... end` [`oc`'s scope is within ...]

This cast creates a temporary variable `v` whose ***static type*** is `A`, and whose ***dynamic type*** is that of `oc`. Since `A` is an ancestor class of `oc`'s ***static type*** (`C`), performing this cast **narrows** the expectations: we can no longer expect `v.c`, but only `v.a` can be expected.

On the other hand, the following cast does **not compile**:

– `check attached {B} oc as v then ... end`

This cast does not compile because `B` is neither a descendant nor an ancestor class of `oc`'s ***static type*** (`C`).

The above example is summarized in Figure 3.

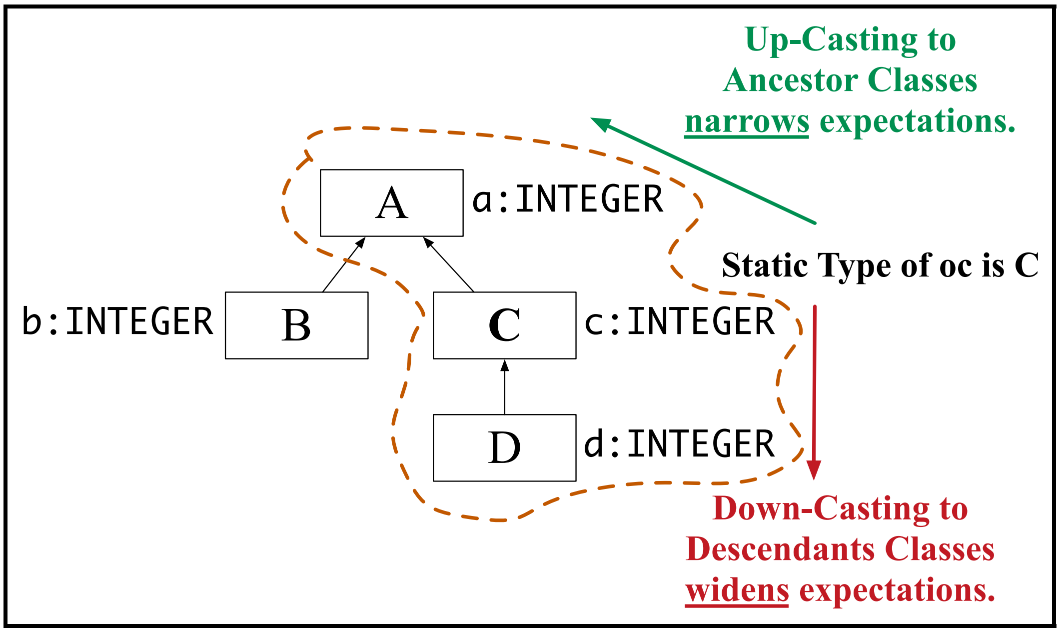

Figure 3: Compilable Casts Given `oc`'s Static Type is `C`

4.2 Does a (Compilable) Cast Cause an Assertion Violation at Runtime?

Consider the following lines of Eiffel code

```
oa: A
create {C} oa.make
```

which declare variable `oa`'s ***static type*** as `A` and initializes its ***dynamic type*** as `C`. According to the principle in Section 4.1, we know that the following casts (where each class being cast into is either a descendant class or an ancestor class of `oa`'s ***static type***, i.e., `A`) are *compilable*:

- `check attached {A} oa as v then ... end`
- `check attached {B} oa as v then ... end`
- `check attached {C} oa as v then ... end`
- `check attached {D} oa as v then ... end`

However, **a cast being compilable does not mean that it will not result in error at runtime.** To determine if there will be a **runtime** error or not, we need to also consider `oa`'s ***dynamic type*** (i.e., `C`):

- `check attached {A} oa as v then ... end`

  This cast works well at runtime.

  ∵ You can use a `C` object as if it were an `A` object. This is because `A` only expects `a`, whereas `C` provides `a` and `c`.

- `check attached {B} oa as v then ... end`

  This cast causes an **assertion violation** at runtime.

  ∵ You cannot use a `C` object as if it were a `B` object. This is because `B` expects both `a` and `b`, but attribute `b` is not declare in class `C`.

- `check attached {C} oa as v then ... end`

  This cast works well at runtime.

  ∵ You can use a `C` object as if it were a `C` object. This is because `C` has the same expectations as itself.

- `check attached {d} oa as v then ... end`

  This cast causes an **assertion violation** at runtime.

  ∵ You cannot use a `C` object as if it were a `D` object. This is because `D` expects both `a`, `c`, and `d`, but attribute `d` is not declare in class `C`.

The above example is summarized in Figure 4.

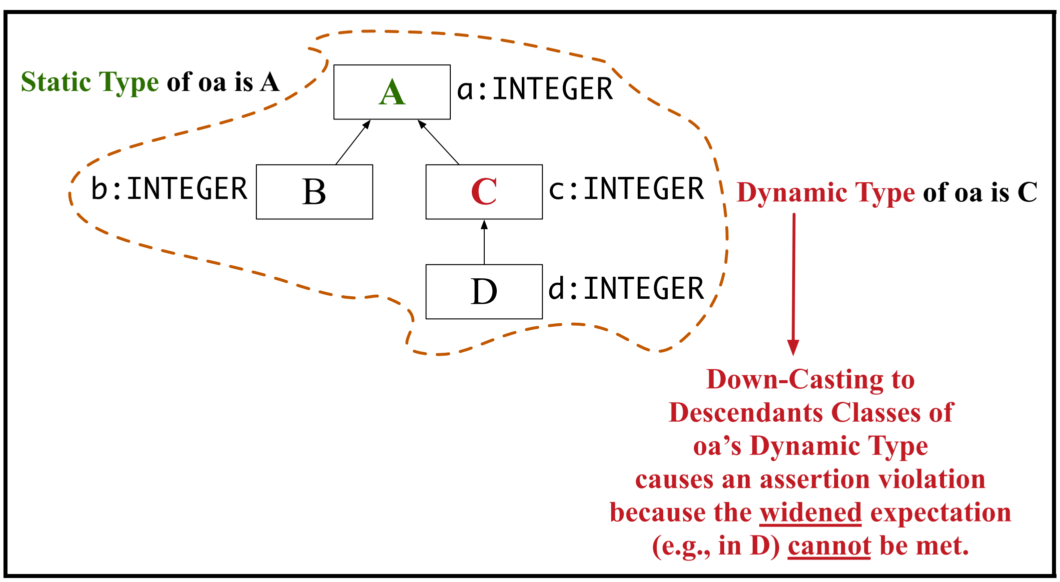

Figure 4: Compilable but Exceptional Casts Given `oa`'s Static Type is `A` and Dynamic Types is `C`

Again, at *runtime* there is an assertion violation resulted from a type cast when the ***dynamic type*** cannot meet the expectations of the reference variable, determined by either its **declared static type** or **temporary static type** resulted from a cast.