EECS2030 Fall 2019
Additional Notes
Static Types, Expectations, Dynamic Types, and Type Casts

Chen-Wei Wang

Contents

1 Inheritance Hierarchy

Consider the following definitions of Java classes

```
class A {
  int a;
  A() {}
}
```

```
class B extends A {
  int b;
  B() {}
}
```

```
class C extends A {
  int c;
  C() {}
}
```

```
class D extends C {
  int d;
  D() {}
}
```

which form the class hierarchy as shown in Figure 1:

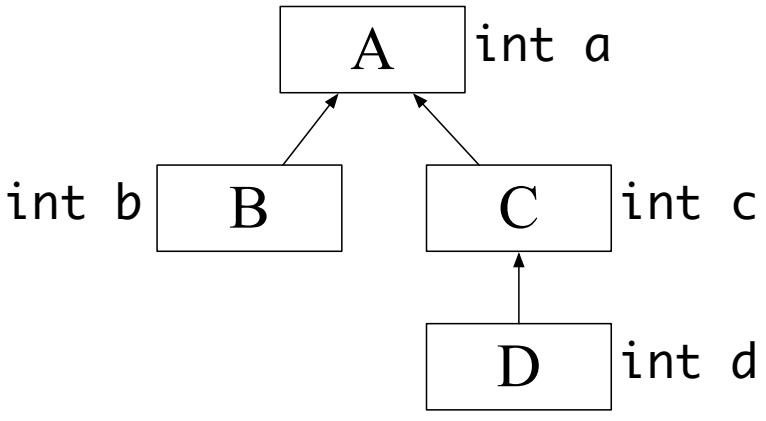

Figure 1: Class Inheritance Hierarchy

2 Static Types Define Expected Usages

Consider the following line of Java code, declaring class `C` as the type of a reference variable `oc`:

```
C oc;
```

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 attributes and methods 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 (i.e., attributes and methods) from its ancestor classes. Therefore, it is **not** expected to call `oc.b` (∵ class `B` is not an ancestor class of `C`), and **not** expected to call `oc.d` (∵ 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
`A oa;`	`oa.a`
`B ob;`	`ob.a` `ob.b`
`C oc;`	`oc.a` `oc.c`

`D od;`	`od.a`
	`od.c`
	`od.d`

Figure 2: Declarations of Static Types and Expectations

3 Dynamic Types

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 assignments are not valid:

```
1 C oc1 = new A();
2 C oc2 = new B();
```

Both of the above assignments are not valid:

- For **Line 1**, 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 2**, 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 assignments are valid:

```
C oc3 = new C();
C oc4 = new D();
```

In the above assignments, 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 Java 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 Java code causes an exception 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 attributes and methods as does that class).
- 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`, the following casts are compilable:

1. `(D) oc`
   Since `D` is a descendant class of `oc`'s static type (`C`), performing this cast **widens** the expectations: we can now expect `((D) oc).d`, whereas `oc.d` cannot be expected.
2. `(C) 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: `((C) oc).a` and `((C) oc).c`.
3. `(A) oc`
   Since `A` is an ancestor class of `oc`'s static type (`C`), performing this cast **narrows** the expectations: we can no longer expect `((A) oc).c`, but only `((A) oc).a` can be expected.

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

- `(B) oc`
  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.

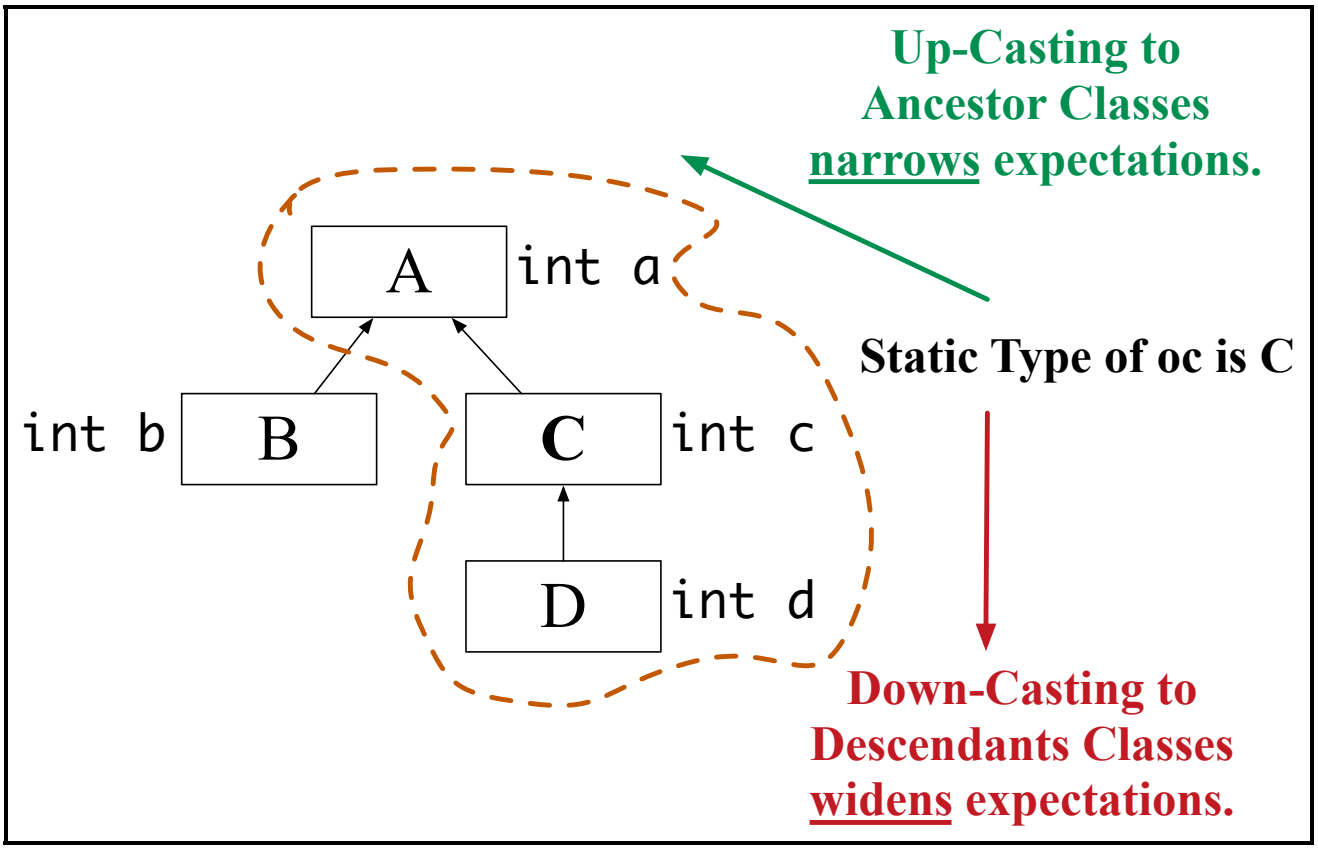

Figure 3: Compilable Casts Given `oc`'s Static Type is `C`

4.2 Does a (Compilable) Cast Cause a `ClassCastException` at Runtime?

Consider the following line of Java code

```
A oa = new C();
```

which declares 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:

- `(A) oa`
- `(B) oa`
- `(C) oa`
- `(D) oa`

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`):

- `(A) oa`

  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`.

- `(B) oa`

  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`.

- `(C) oa`

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

- `(D) oa`

  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.

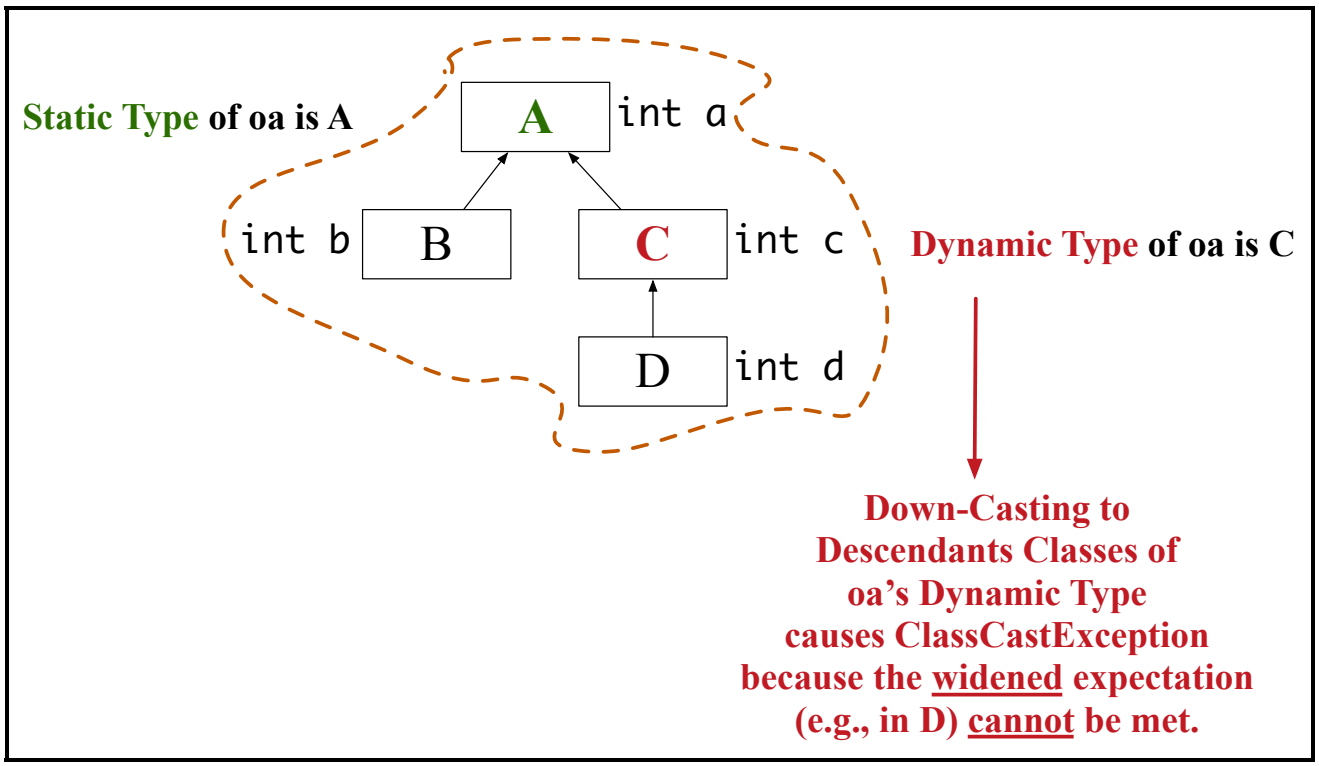

Figure 4: Compilable but Exceptional Casts Given `oa`'s Static Type is `A` and Dynamic Types is `C`

Again, at runtime there is a `ClassCastException` 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.