Apparent and real classes

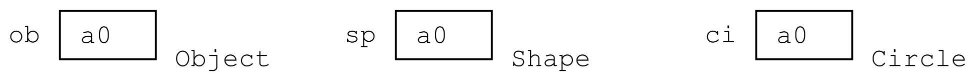
The purpose of any terminology is to facilitate discussions of issues and clarify concepts. Without the terminology, discussions may be long and laborious and confusing. With the terminology, the issues and concepts should be explained and discussed in shorter time. In this blecture, we introduce two new words: the *apparent class* and *real class* of a variable or expression. Please make sure you understand this new terminology.

The apparent class of a variable

Here are basic declarations of three variables: `ob`, `sp`, and `ci`.

```
Object ob;   Shape sp;   Circle ci;
```

And here are the variables themselves. We have annotated them with their types. They all contain the name of the same object, of class Circle.



In this situation, we say that the *apparent class* of `ob` is `Object`, the *apparent class* of `sp` is `Shape`, and the apparent class of `ci` is `Circle`:

Definition: The *apparent class of a variable* is the class with which it is declared.

Why do we use this name? *Apparent* means clearly *seen or understood*, *appearing to show particular qualities or attributes that may not be genuine*. Apparently, the variable contains an object of the given class.

The apparent class is a *syntactic property*. You can tell from the program itself, without executing it, what the apparent class of a variable is. And the apparent class determines what can and cannot be referenced.

In the situation shown here, since `ob`'s apparent class is `Object`, only fields and methods that are defined in class `Object` can be referenced using `ob`. Those that cannot be referenced have been grayed out. Since `sp`'s apparent type is `Shape`, only fields and methods that are defined or inherited in class `Shape` can be referenced using `sp`. Those that cannot be referenced have been grayed out. And, since `ci`'s apparent class is `Circle`, only fields and methods that are defined or inherited in class `Circle` can be referenced using `ci`.

In summary, the rule is:

> Rule: For a variable `x` of some class-type `C`, the legal references of the form `x`.*variable* or `x`.*method-call* are to variables and methods defined in or inherited by class `C`.

The real class of a variable

The apparent class is what the object in a variable appears to be from its declaration; the real class is what the object in the variable really is. At the moment, the apparent class of `ob` is `Object`, but the real class of `ob` is `Circle`, because `ob` contains the name of an object that is of class `Circle`.

This real class can change during execution, whenever an assignment to the variable is executed.

The real class determines which method is actually called. For example, in the situation given here, `ob.toString()` is legal, because `toString` is defined in class `Object`. However, it calls the overriding `toString` function that is declared in class `Circle` —this is a consequence of the bottom-up rule for finding the declaration in an object corresponding to a variable or method.

The fact that the overriding `toString` function is called may seem strange at first. But it is an important aspect of OO programming. Here's how to see that it makes sense. When the call `ob.toString()` is evaluated, we would like as much information as possible about the object. If function `toString` in the `Object` partition were called, all we would get is the name of the object! But, since the `toString` function in partition `Circle` is called, we get the maximum amount of information.